PyDVE: An Open-Source Python-Based Design Verification Framework for RTL Emulation

Curtis Bucher
March 2024
Computer Engineering, Cal Poly San Luis Obispo

1 INTRODUCTION

In the digital hardware development field, ensuring the functionality of a design is crucial before moving to fabrication. This process of functional design verification detects and rectifies any issues prior to manufacturing, thereby reducing the risk of costly errors. However, this verification process is both complex and resource-intensive, consuming an estimated 70% of time and resources in the digital design process and creating a bottleneck on the speed that new products can be brought to market [1]. As hardware complexity continues to grow exponentially, finding efficient and cost-effective verification methods becomes increasingly important.

To address this challenge, there is a broad effort to develop tools and frameworks that streamline the verification process. Broadly, these systems fall into two categories: simulation and emulation. I will further draw a distinction between tools that are available from traditional "Big 3" EDA vendors, `Cadence`, `Synopsis`, and `Siemens`, with newer, open-source tools that are beginning to gain traction in the functional verification landscape.

This paper introduces a novel verification framework that strives to combine the speed of traditional emulation tools with the power, ease, and flexibility of the `Python` ecosystem. Notably, this framework is open-source and utilizes inexpensive off-the-shelf hardware, eliminating the high costs associated with traditional emulation platforms.

The paper also explores the advantages of this framework, highlighting its flexibility and accessibility compared to conventional emulators. Additionally, potential performance improvements over existing simulation tools are investigated.

2 BACKGROUND

Broadly, functional design verification deals with two questions: are two models of the design under verification equivalent, and does a model of the design operate correctly [2]? This is analogous to writing unit tests in software engineering. To answer these questions for a hardware design, it is necessary to replicate the behavior of the design quickly and accurately, to see how it behaves under different test stimulus. To reproduce the behavior of the hardware model, two main methods are used: simulation and emulation. Each of these methods have different tools available from the "Big 3" vendors, as well as open-source alternatives. We will explore the benefits and drawbacks of each method in depth.

These models are described using a traditional hardware description language (HDL) such as `SystemVerilog` or `VHDL`. Additionally, module testbenches, used to drive input stimuli to the design and verify its output accuracy, are traditionally written in `SystemVerilog`, often utilizing `Universal Verification Methodology` (`UVM`) [1]. However, the increasing success of open-source verification tools has introduced new languages for testbench design, with `Python` emerging

as a powerful, open-source alternative to traditional *SystemVerilog* testbench design, which we'll delve into further.

Simulation

Simulation is the most common form of design verification [2]. High performance simulators are available from most of the major EDA vendors, including *Siemens ModelSim*, *Synopsis VCS*, and *Xilinx Xsim* [3] [4] [5]. These programs run on a traditional computer to simulate the behavior of RTL hardware models. The objective with simulation is to enable quick and flexible development of these testbenches.

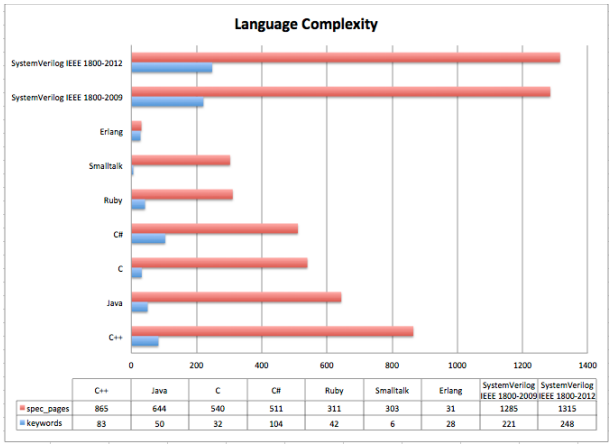
Emulation

Emulation, on the other hand, utilizes a dedicated hardware emulator to mimic the behavior of the RTL model without the need for compilation and execution on a conventional computer. Although mapping an RTL design onto emulation hardware requires significant testbench overhead, the hardware enables individual test vectors to run thousands of times faster, a desirable tradeoff for larger designs with complex testbenches [2].

In most setups, a host computer manages the testbench and generates the test vectors to keep the emulator occupied. Emulation tools and the necessary hardware are available from major EDA vendors like *Cadence Palladium Z1*, *Synopsys ZeBu*, *Siemens Veloce*, and *Xilinx Alveo* [6] [4] [3] [5]. However, these software tools and accompanying emulation hardware are often prohibitively expensive, posing a barrier to entry for verification engineers seeking to leverage the benefits of emulation.

Python

Python, as an open-source, general-purpose programming language, has gained traction in testbench design alongside the increasing availability of open-source design verification tools, which we'll explore later in more detail. *Python* testbenches offer advantages due to the language's flexibility and ease of use compared to a traditional HDL like *SystemVerilog* or *UVM*. *Python* is familiar to a larger pool of engineers, and its ecosystem provides access to numerous packages for developing rich, powerful testbenches across a variety of domains [7]. In terms of language complexity, *Python* is notably simpler, with only 23 keywords compared to *SystemVerilog*'s 221 keywords [8].



	C++	Java	C	C#	Ruby	Smalltalk	Erlang	SystemVerilog IEEE 1800-2009	SystemVerilog IEEE 1800-2012
spec_pages	865	644	540	511	311	303	31	1285	1315
keywords	83	50	32	104	42	6	28	221	248

Figure 1: http://www.fivecomputers.com/language-specification-length.html

3 RELATED WORK

Due to the expensive, closed source nature of the verification tooling from the big vendors, and the success of the open-source software ecosystem, there has been a substantial effort in recent years to develop a suite of open-source tools for digital hardware design, including a host of open-source *Verilog* simulators and emulators. Some of these open-source simulation tools have *Python* front-ends available, exposing the verification engineer to the flexibility and power of the *Python* ecosystem.

Tool	Open Source	Simulation	Emulation	Python-based
Traditional Tools		X	X	
Icarus Verilog	X	X		
Cocotb	X	X		X
Verilator	X	X		
PyMTL	X	X		X
Firesim	X		X	

Table 1: Summary of Open-Source Verification Tools

Icarus Verilog

Icarus Verilog is a free and open-source *Verilog* compiler, intended to generate high performance code for back-end tools [9]. Although in development since 2000, it still exhibits notable limitations compared to other simulators. Notably, *Icarus Verilog* lacks support for *SystemVerilog* or *UVM* testbenches, which are the modern standard for complex testbench design, limiting its utility within the industry.

Cocotb

Cocotb is another free and open-source digital logic verification framework, notable because *cocotb* testbenches are written using the open-source *Python* programming language, rather than *SystemVerilog*. *Cocotb* is not another simulator, but rather a *Python* frontend that supports many existing simulators, including open-source solutions like *Icarus Verilog*, as well as closed source alternatives like *Synopsys VCS* and *Siemens Modelism*. According to the documentation, cocotb encourages the same philosophy of design reuse and randomized testing as *UVM* but implemented in *Python* [7].

```python
import cocotb
from cocotb.triggers import RisingEdge, FallingEdge

class SimpleDriver:

    def __init__(self, dut):
        self.dut = dut
        self.value = 0

    @cocotb.coroutine
    def drive(self):
        while True:
            yield RisingEdge(self.dut.clk)
            self.dut.data <= self.value

@cocotb.test()
def test(dut):
    driver = SimpleDriver(dut)
    cocotb.fork(driver.drive())
    yield FallingEdge(dut.clk)
    driver.value = 1
```

Figure 2: Sample cocotb code [8]

Verilator

Verilator claims to be the fastest *Verilog*/*SystemVerilog* simulator available, outperforming many closed source commercial simulators by 200-1000x [10]. It achieves this by compiling the *Verilog* or *SystemVerilog* module into an optimized, multithreaded model of the design, which is then wrapped in a *C++* or *SystemC* wrapper and compiled using traditional compilers like *GCC* [10].

Despite its impressive performance and support for both *SystemVerilog* and *Verilog*, *Verilator* lacks many features of traditional simulators and is primarily intended for high-speed simulation and integration of *SystemVerilog* models with *C* code [10]. Furthermore, it does not natively support *Python* testbenches and is not a supported backend of *cocotb*, limiting verification engineers to *Verilog* and *SystemVerilog* testbenches.

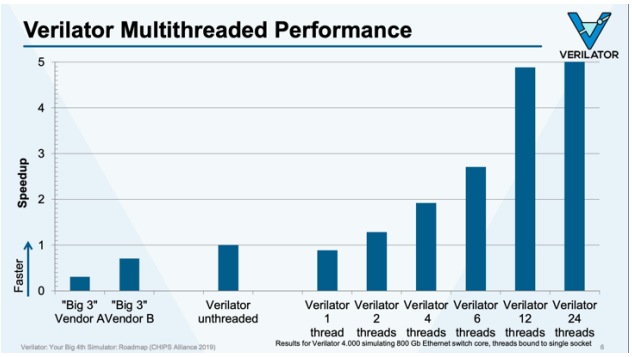


Figure 3: Verilator Multithreaded Performance [11]

PyMTL

PyMTL is a novel, free and open-source *Python*-based hardware generation, simulation, and verification framework [12]. In the *PyMTL* workflow, hardware designs and their corresponding testbenches are both defined using *Python*, gradually refined by the verification engineer. Once the design is finalized, the *PyMTL* framework facilitates exporting the testbench and model as *Verilog* code, compatible with traditional EDA tools.

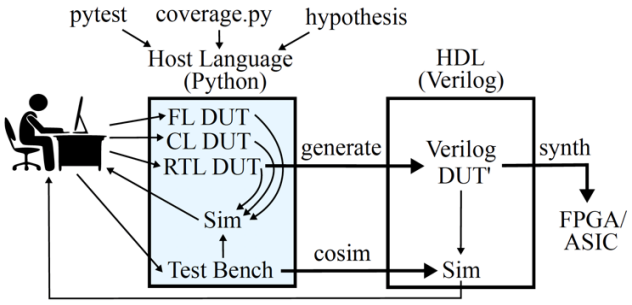


Figure 4: PyMTL's workflow.

Firesim

`Firesim` serves as an open-source emulation alternative from UC Berkeley, utilizing general-purpose FPGAs to accelerate the simulation process either locally or in the cloud using Amazon F1 instances. Testbenches within `Firesim` can be written using traditional `Verilog` or `Chisel`, Berkeley's hardware description language, and emulated at speeds ranging from 10s to 100s of MHz [13]. However, while `Firesim` represents a significant advancement in the open-source hardware movement, it is limited to testbenches written in `Verilog` or `Chisel`, thus precluding verification engineers from leveraging the powerful `Python` ecosystem in their testbench designs.

4 DEVELOPMENT

PyDVE

Considering the popularity of `Python`-based simulation frameworks, and the remarkable speedup of emulation offerings from traditional vendors, there is a notable lack of solutions for running `Python`-based testbenches with emulation. `PyDVE` was developed to bridge this gap, offering an open-source framework for running `Python`-based testbenches with emulation. The goal of `PyDVE` is to combine the flexibility, ease-of-use, and extendibility of the `Python` ecosystem with the speed benefits of a traditional emulation system.

	SYSTEMVERILOG	**PYTHON**
SIMULATION	Traditional Simulators, Icarus Verilog	Cocotb
EMULATION	Traditional Emulators, Firesim	**PyDVE**

Table 2: The missing python-based emulation tool...

Architecture

Since `PyDVE` relies on emulation to replicate the behavior of the design under test (DUT), additional hardware is necessary for its operation. To ensure affordability and accessibility, a hardware architecture utilizing readily available components was chosen. Specifically, the `Kria KV260` development board by `Xilinx`, priced under $400, was selected. This board features the `Zynq UltraScale+ MPSoC`, integrating an ARM processor tightly coupled with a `Zynq` FPGA.

The choice of the `Kria` board is rooted in its support for the PYNQ framework, an open-source project developed by AMD, which provides a `Python` software interface for interacting with FPGA designs [14]. This compatibility ensures seamless integration with the `PyDVE` package, facilitating efficient testing processes.

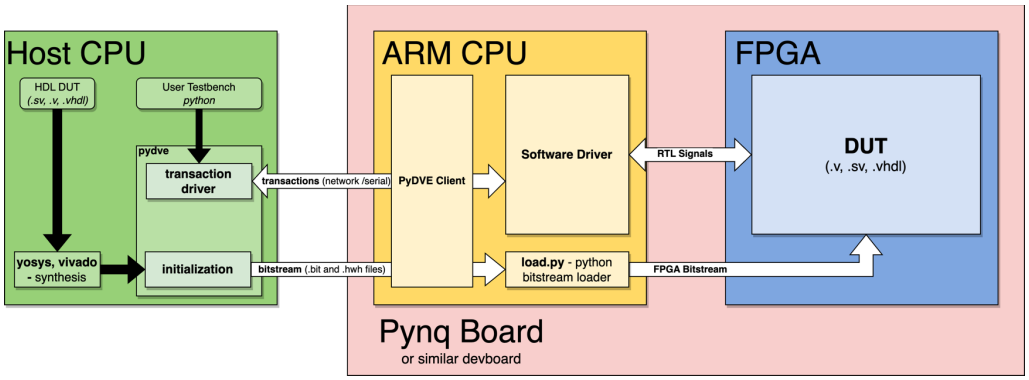


Figure 5: PyDVE Hardware and Software Architecture

The system consists of a host computer connected to the `Kria` board via either a high-speed network or a serial connection, enabling multiple `Kria` boards to connect simultaneously for concurrent testing and a greater test throughput.

On the host CPU, engineers access the `PyDVE Python` package, facilitating the transmission of the DUT to the development board and the transmission of transaction-level stimulus to the emulator board.

Meanwhile, the `Kria` Board's embedded CPU operates a server, which awaits commands from the host computer. This server manages interactions with the FPGA, including loading the DUT, providing stimulus to the design, and retrieving outputs from the design.

The FPGA assumes responsibility for emulating the DUT by loading it into its programmable logic and communicating with the embedded server via high-speed AXI GPIO. Its flexibility allows it to emulate multiple DUTs simultaneously, further enhancing system speed.

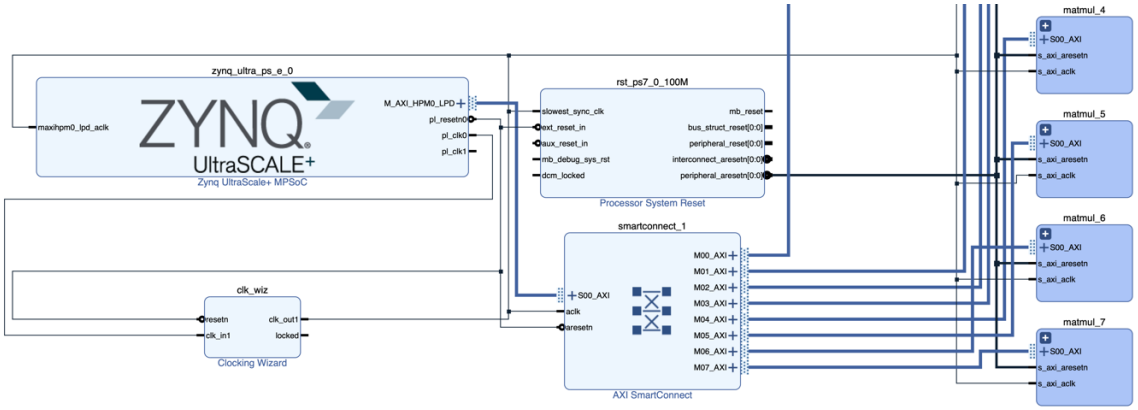


Figure 6: PyDVE FPGA Design with 4 of 7 DUTs

Design Decisions

Several hardware and software architectures were explored before arriving at the final architecture depicted above. One key consideration was determining where to run the testbench—on the host CPU or the embedded CPU on the `Kria` board. Ultimately, running the testbench on the host system was chosen to reduce demands on the embedded hardware, potentially improving

system speed and lowering costs. While this approach introduces some network delay, it was deemed a necessary tradeoff to achieve the desired performance. Additionally, utilizing a synthesizable testbench directly on the FPGA was considered but dismissed due to concerns about restricting the system's flexibility by requiring testbenches to be written entirely in a limited subset of synthesizable *Python*.

Verification Process

The *PyDVE* framework is presented to the verification engineer as a *Python* package, featuring a class called *PyDVE* for interacting with the DUT. This framework enables engineers to write verification testbenches within traditional *Python* software testing frameworks like *unittest* or *pytest*, interacting with the DUT as if it were a standard *Python* class.

The typical verification workflow with *PyDVE* involves several steps:

Synthesizing the Design – Currently, no open-source tools exist for generating bitstreams for *Xilinx* FPGAs from HDL code. Hence, engineers are responsible for initially synthesizing their design using traditional synthesis tools like *Xilinx Vivado*. Once synthesized, the DUT is exported as a bitstream. Future work on the project will focus on integrating existing synthesis tools within the framework.

Loading the Design - Next, engineers provide the *PyDVE* testing framework with the DUT, represented by the exported files containing synthesized *SystemVerilog* code and necessary metadata. These files are loaded onto the FPGA through the *PyDVE* package's `sources` decorator, which facilitates the transmission of the bitstream over the network connection to the embedded server for loading onto the FPGA.

```python
@pydve.sources("dut.bit", "dut.hwh", always_reset=False)
class TestArithmetic():
    def testAdd(self, a, b):
        assert self.dut
        # calculating flags
        flags = 0b000
        # zero
        if a + b == 0:
            flags |= 0b001
        # sign
        if (a + b) & 0x80000000:
            flags |= 0b010
        # overflow
        if (a & 0x80000000) == (b & 0x80000000) and
            (a & 0x80000000) != (a + b & 0x80000000):
            flags |= 0b100
        # driving transaction
        self.dut.operandA @= a # type: ignore
        self.dut.operandB @= b # type: ignore
        self.dut.opcode <<= ADD # type: ignore

        # Testing
        assert self.dut.result == a + b & 0xffffffff
        assert self.dut.flags == flags
```

Figure 7: PyDVE code snippet of an ALU addition test.

Implementing and Running the Testbench – As mentioned prior, the *PyDVE* framework seamlessly integrates with existing *Python* testing frameworks like *pytest* and *unittest*. Running

and developing *Python* testbenches should be familiar to engineers with a working knowledge of *Python*. Testbenches can make use of *Python*'s extensive ecosystem of libraries for powerful, high-level testing. The simplicity of interacting with the FPGA and DUT via a *Python* class abstraction allows engineers to write testbenches quickly and easily, with minimal understanding of the underlying system.

In the code snippet above, `testAdd()` is an individual test vector within the `TestArithmetic` testbench, responsible for testing that an ALU correctly performed an addition operation. Within this test, `self.dut` is an object of type `pydve.pydve`. The `self.dut` object exposes all the input and output ports of the devices-under-test. Input ports are driven by the verification engineer through the `@=` and `<<=` operators, which assign values to the input ports combinatorically and sequentially. This can be seen on lines 18-20. Whenever the input ports of the `self.dut` are assigned to, the values are collected and driven to the embedded server in the form of a high-level transaction, which is then driven to the FPGA as individual RTL signals. Every time the DUT is updated on the FPGA, the resultant output signals are read by the embedded server as RTL signals, which are then collected into a high-level transaction and sent back to the host computer. The verification engineer can then check the validity of these output signals within the testbench, by accessing the output ports of the `self.dut` object. This can be seen on lines 23-24.

By abstracting the complexity of interacting with the FPGA and DUT into a *Python* class, *PyDVE* enables the rapid development of *Python* testbenches, leveraging the language's extensive ecosystem for powerful testing capabilities. Examples in the source code demonstrate the use of *Python* testbenches, including leveraging the *numpy* package for checking the performance of a matrix multiplier module and utilizing the *hypothesis* package for generating constrained random stimulus.

5 EVALUATION

To assess the performance of the system, *PyDVE* was compared against traditional simulation tools, as well as open-source alternatives, using either *Python* and *SystemVerilog* testbenches depending on the tool. These evaluations were conducted on a matrix multiplier module obtained from *cocotb's* source code as a motivating example, varying the number of test vectors from 100 to 100,000 cases. This approach provides insights into the runtime of each tool and how their performance scales with longer testbenches. The following graphs depict the total testbench runtime and test vector throughput for *PyDVE* and various simulation tools. Notably, the data do not include synthesis times for the modules from *Vivado*. Detailed data and methodology can be found in tables 3 and 4 in the appendix.

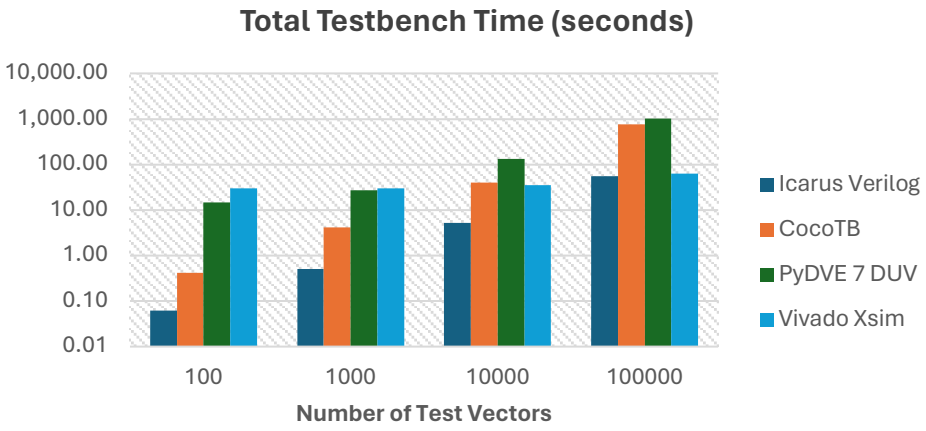


Figure 8: Comparison of Total Testbench Time for Different Simulators. See Table 3 in Appendix.

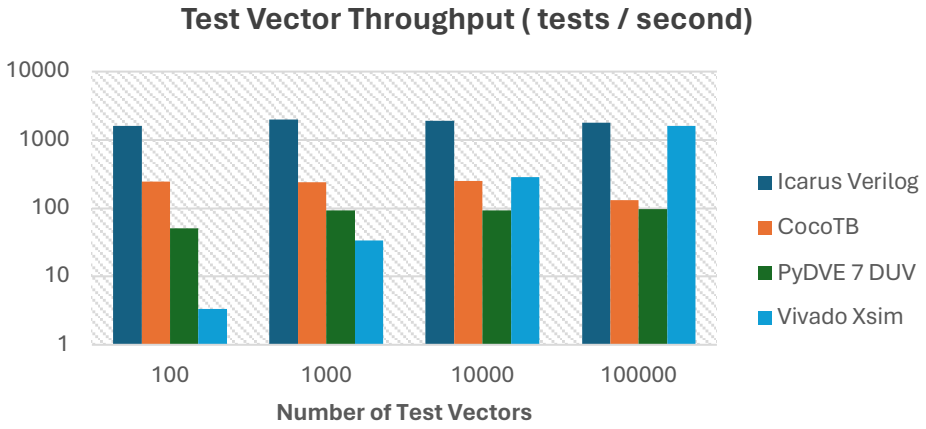


Figure 9: Comparison of Test Vector Throughput for Different Simulators. See Table 4 in Appendix.

Initial Performance Evaluation

During testing, the initial version of `PyDVE`, operating in parallel with 7 DUTs, exhibited slower performance compared to `Icarus Verilog` and `cocotb` across all scenarios. Additionally, `PyDVE` lagged behind `Vivado Xsim` for 10,000 and 100,000 test vectors. This trend is also evident in the test vector throughput analysis, where `PyDVE` underperformed relative to `cocotb` and `Icarus Verilog` and exhibited inferior scalability compared to `Vivado Xsim` with higher numbers of test vectors. These findings suggest potential room for performance improvements, which will be further discussed in subsequent sections.

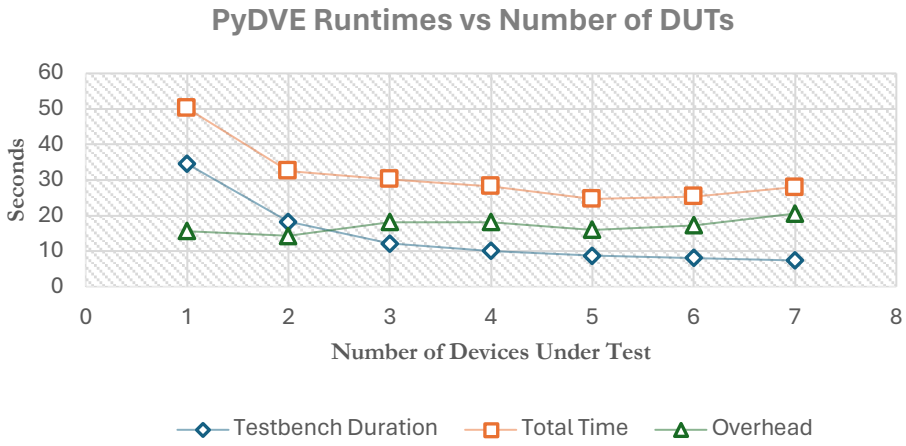


Figure 10: PyDVE Runtimes Vs. Number of Devices Under Test. See Table 5 in Appendix.

Performance Scaling with Parallelism

A study of *PyDVE's* performance with varying levels of parallelism reveals insights into its scalability. As discussed in the "Architecture" section, *PyDVE* supports testing parallelism with multiple DUTs on the same FPGA. Increasing the number of DUTs leads to a decrease in total testbench time due to enhanced test vector throughput with greater parallelization. However, there is a corresponding rise in *PyDVE's* overhead as the system must initialize and manage more devices. Consequently, the system achieves peak performance when operating with 5 devices under test, striking a balance between the performance overhead of managing multiple devices and the throughput gains from increased parallelization. It's worth noting that the *Kria* system encountered limitations when managing more than seven devices, indicating potential opportunities for performance enhancements in the design of the embedded hardware server.

6 CONCLUSION

While *PyDVE* did not immediately surpass other simulation methods as anticipated, I remain optimistic about its potential based on the promising results observed. As highlighted in the evaluation section, several bottlenecks significantly hinder the overall system speed, particularly within the embedded server on the *Kria* platform. The server, developed in *Python* for rapid prototyping, may not be the most efficient option. I believe that transitioning to a low-level language tailored to the hardware constraints of the *Kria* platform could better leverage the FPGA's capabilities and enhance test vector throughput. Moreover, while supporting off-the-shelf hardware is essential, there's an opportunity to design specialized hardware for the framework, unrestricted by the limitations of the *Kria* platform.

In addition to addressing performance concerns, I intend to streamline the testbench development process by integrating *PyDVE* with other verification software. Plans include integrating synthesis tools like *Xilinx Vivado* or *Yosys* into the platform for automated bitstream generation. This would simplify the workflow for verification engineers, allowing them to provide their HDL code to the system without the need for separate bitstream compilation. Furthermore, future versions of *PyDVE* aim to support running *cocotb* testbenches, creating a unified environment for executing *Python* testbenches in both emulation and simulation. Such flexibility would empower engineers to fully capitalize on the benefits of both emulation and simulation.

Throughout the development of *PyDVE*, I gained valuable insights into design verification tools, particularly open-source options. I firmly believe that nurturing the growth of open-source tools is pivotal for the future of digital hardware design. These tools introduce competition and innovation to an industry dominated by a few major vendors. *Cocotb*, *Firesim*, and *PyDVE* hold the potential to enhance efficiency in digital design and verification, crucial as designs continue to grow in complexity.

7 BIBLIOGRAPHY

[1] A. Molina, "Functional Verification: Approaches and Challenges," *Latin American Applied Research,* vol. 37, no. 1, 2007.

[2] Synopsis Inc, "Functional Verification 2003: Technology, Tools and Methodology," IEEE Xplore, 2003.

[3] Siemens, "Verification and Validation," 2024. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/verification-and-validation/. [Accessed 11 03 2024].

[4] Synopsis, "Verification Family," 2024. [Online]. Available: https://www.synopsys.com/verification.html. [Accessed 11 03 2024].

[5] Advanced Micro Devices, "Vivado Overview," 2024. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html. [Accessed 11 03 2024].

[6] Cadence, "Cadence Verifcation," 2024. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification.html. [Accessed 11 03 2024].

[7] Cocotb, "Cocotb Documentation," 2023. [Online]. Available: https://docs.cocotb.org/en/stable/. [Accessed 09 03 2024].

[8] B. Rosser, "CocotbL A Python-based digital logic verification framework," University of Pennsylvania.

[9] S. Williams, "Icarus Verilog," 2019. [Online]. Available: https://steveicarus.github.io/iverilog/. [Accessed 09 03 2024].

[10] Veripool, "Welcome to Verilator," Veripool, 2024. [Online]. Available: https://veripool.org/verilator/. [Accessed 09 03 2024].

[11] W. Snyder, "Verilator: Your Big 4th Simulator: 2019 Intro and Roadmap," Chips Alliance, 2019.

[12] S. Jiang, C. Torng and C. Batten, "An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework," in *Workshop on Open-Source EDA Technology*, Ithaca, NY, 2018.

[13] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach and K. Asanovic, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *ACM/IEEE International Symposium on Computer Architecture*, Berkeley, CA, 2018.

[14] Advanced Micro Devices Inc, "PYNQ: Python productivity for Adaptive Computing platforms," 2022. [Online]. Available: https://pynq.readthedocs.io/en/latest/#. [Accessed 11 03 2024].

8 APPENDIX

Performance Comparison (Figure 8,9)

Performance tests in tables 3 and 4 were run on `Intel Core-i7` with 32 GB RAM.
`Vivado` synthesis for 7-DUTs took 24 minutes.

Num Tests	Icarus Verilog	Cocotb	PyDVE 7 DUTs	Vivado Xsim
100	0.06	0.41	14.65	30.00
1000	0.51	4.15	27.25	30.00
10000	5.22	40.19	131.79	35.00
100000	55.52	760.49	1027.46	63.00

Table 3: Total Testbench Time (seconds) vs. Number of Test Vectors.

Num Tests	Icarus Verilog	Cocotb	PyDVE 7 DUTs	Vivado Xsim
100	1612.90	243.90	50.34	3.33
1000	1972.39	240.96	92.12	33.33
10000	1914.98	248.82	92.35	285.71
100000	1801.15	131.49	97.33	1587.30

Table 4: Test Vector Throughput (tests/second) vs. Number of Test Vectors.

Parallelism Assessment (Figure 10)

Parallelism assessments in table 5 were run on `Apple M1` with 16 GB RAM.
`Vivado` synthesis for 7-DUTs took 24 minutes on `Intel Core-i7` with 32 GB RAM.

Num DUVs	Testbench Duration (s)	Total Time (s)	Overhead (s)
1	34.5515757	50.1891054	15.63752971
2	18.2105112	32.5236315	14.31312038
3	12.1188674	30.2632	18.14433262
4	10.0755533	28.2098582	18.13430492
5	8.71449667	24.6855627	15.97106604
6	8.11689833	25.3971535	17.28025513
7	7.3822885	27.9552509	20.57296242

Table 5: PyDVE Timing Breakdown (s) per Number of Test Vectors.