2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

The Cost of Standard Verification Methodology Implementations

Abigail Williams, Graphcore Ltd, Bristol, UK (abigailw@graphcore.ai)
Svetlomir Hristozkov, Graphcore Ltd, Bristol, UK (svetlomirh@graphcore.ai)
Adam Hizzey, Graphcore Ltd, Bristol, UK (adamh@graphcore.ai)

***Abstract*—The silicon verification industry typically uses a combination of the Universal Verification Method and SystemVerilog based constrained random metrics-driven approach to verification. This paper will quantify and contrast the performance costs of this approach for frequent actions across multiple EDA vendors and against a proprietary C++ implementations. The setup and measures taken in ensuring the fairness of these microbenchmarks are presented. Finally, results are compared over time to ensure reliability of the data and conclusions drawn.**

Keywords— SystemVerilog; C++ ; Performance Measurement; Microbenchmarks; String Formatting;

I. INTRODUCTION

The verification industry has largely agreed on its approach to verification – employing a constrained-random and metrics-driven strategy [1]. While some variation exists (e.g. CocoTb [2]), many projects favour a testbench which uses the Universal Verification Methodology (UVM) for all stimulus generation and SystemVerilog standard coverage features for coverage collection.

The standardization of verification testbench environments has brought about numerous benefits, particularly at the time when the deceleration in Moore’s Law was first introducing a transition to more complex architectures and therefore more simulation cycles. This process has only accelerated since UVM’s IEEE standardization in 2017. Despite this, there is little quantified research into the cost of this implementational choice.

Graphcore’s verification team conversely uses a proprietary in-house implementation of the same concepts but over time has had to integrate it with a variety of UVM/standard SV verification IP. As a result, we have anecdotally observed degraded performance in terms of the number of tests we can run overnight.

Figure 1 provides a visual representation of the proportion of time spent performing certain actions in two different testbench simulations. The testbench used to produce Figure 1a incorporates UVM and SystemVerilog into its verification environment, as well as Graphcore’s inhouse C++ and Python methods for modelling and coverage collection. The testbench used to produce Figure 1b exclusively uses a custom C++ based methodology for verification. Both testbenches use some C++, so a similar proportion of time is spent performing Direct Programming Interface (DPI) calls. The UVM based testbench reports verification time used under the “Verilog Package” category. In a perfect world the verification portion of a testbench will be minimised and the time will be spent actually simulating cycles i.e. under the “Kernel” and “Verilog Module” categories. The Verilog package takes up 5.84% of the simulation time for the first testbench, but just 0.55% for the second. The kernel and Verilog module take up a combined proportion of 82% of simulation time for the first testbench, which is significantly lower than the 91% combined proportion seen in the second testbench. While this is not a perfect apples-to-apples comparison, these figures suggest that there may be an inherent cost involved in using standard UVM based TB with SV based coverage collection.

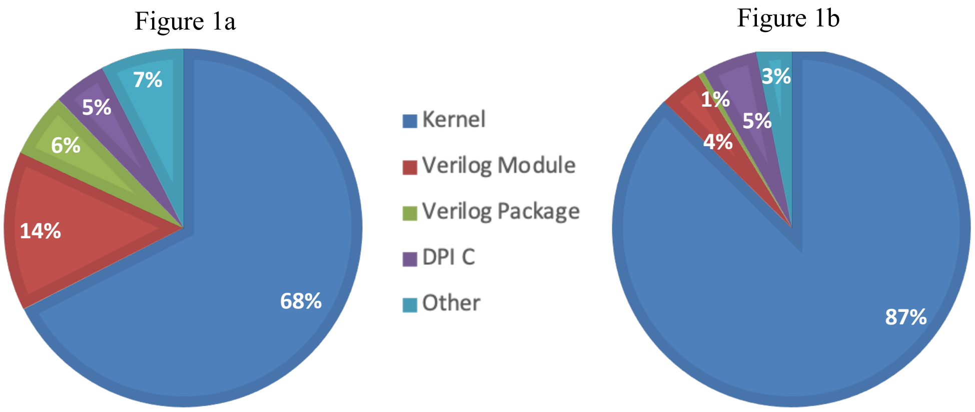

Figure 1 Pie charts displaying the proportion of simulation time spent performing certain actions for two different testbenches. The testbench used to produce Figure a incorporates UVM and SystemVerilog in the verification environment, as well as Graphcore's inhouse verification methods. The testbench used to produce Figure b exclusively utilizes C++ and Python methods of verification.

This paper will present a study into the cost of performing some of the most common actions employed in a verification environment such as this; The selected actions will be evaluated across EDA tool vendors Graphcore had access to at the time of writing and further as well as against Graphcore's inhouse proprietary implementations employing C++ and Python.

The actions selected to be benchmarked are:

- Message printing and formatting
- Coverage collection
- The implementation of the common observer pattern employed in UVM's Transaction Layer Modelling (TLM) vs a proprietary Graphcore inhouse implementation

This choice is based on qualitative judgement and anecdotal evidence when exploring profiling logs generated by EDA tools but also somewhat limited due to the time available for the project. Further exploration into constraint solvers in EDA tools, among others, should be made.

II. Performance Measurement Approach

A. *Measurement Setup Architecture*

In order to compare the performance of a SystemVerilog / UVM verification environment with Graphcore's corresponding implementation, the cost of various commonly performed actions has been quantified. Each of the abovementioned selected actions is implemented using SystemVerilog, then replicated in C++ and executed across all available simulators. The approaches used to measure the time taken for each action to complete are illustrated in Figure 2.

The *Chrono* C++ library [3] is used to measure the elapsed time between the invocation of "*start_measurement"* and the call to *"stop_measurement"*. As seen in Figure 2, each of the actions start in SystemVerilog. When benchmarking SystemVerilog and UVM models, the DPI is used to call *"start_measurement"* and *"stop_measurement"*. However, when benchmarking C++ models, the DPI is used to select the relevant test, with calls to start and stop the timing measurements then being invoked from within C++. Measurements omitting the second step of performing an action have been taken to quantify the time taken for a DPI call; this allows the cost of the additional DPI calls in the SystemVerilog and UVM benchmark tests to be accounted for when contrasting the timing measurements with the equivalent C++ tests.

The benchmark tests are run nightly on a single reserved machine that will have no other jobs running on it, as part of a suite of performance measurements that Graphcore runs on its codebase. The results plots in Section III show the benchmark time of individual test runs (Y-axis) over the days that the tests were run (X-axis). A lower execution time is better.

Only one action is tested at a given time, in order to minimize any disturbances which might interfere with the measurements taken. Each measurement is also performed 100 times, so as to minimize the error in reporting and across each simulator tool due to intermittent machine fluctuations such as OS processes or cosmic rays. Measuring the performance of each action over time will also allow the amount of variation in the measurements to be determined.

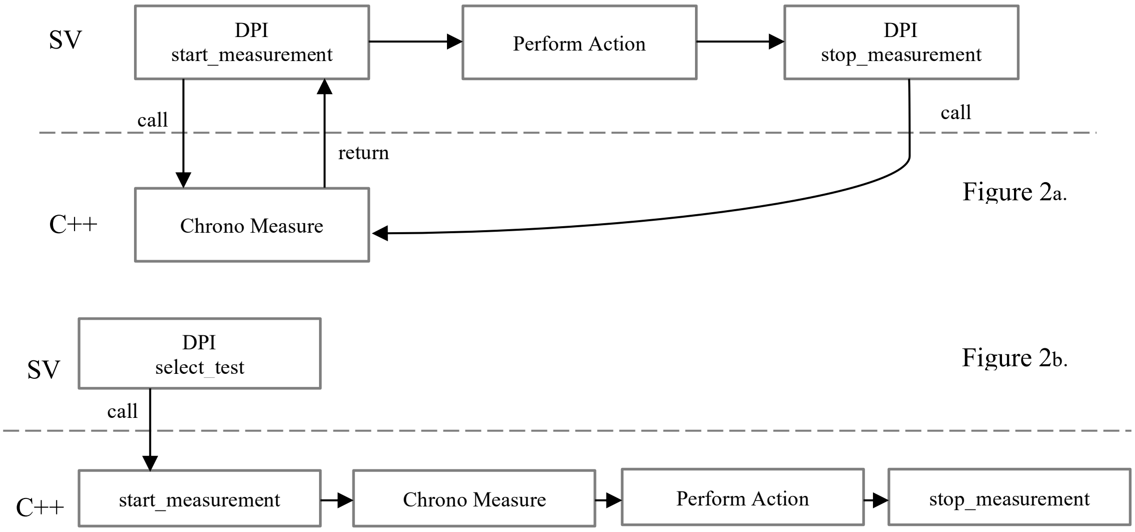

Figure 2 A visualisation of the approach taken to measure the time cost of performing various actions. Figure a shows the method used to benchmark actions in a UVM and SystemVerilog environment. Figure b shows the method used to benchmark the same actions when implemented in a C++ environment.

B. Benchmark Method

Some of the actions likely to incur a significant cost in simulation runtime have been identified as: transaction-level modelling (TLM), coverage collection, formatting and messaging. Two common simulators were tested.

1) **Transaction Level Modelling**

The time cost of TLM refers to the time penalty incurred when transporting a packet from a monitor to the eventual consumer within a TLM network. The model used to benchmark this consists of 100 sequentially connected ports and exports. The timing measurement is started as a uvm_item *containing a single* 32-bit variable is pushed into the first port and then stopped when the same packet reaches the final export in the chain.

2) **Functional Coverage Collection**

The timing measurements taken for coverage collection refer to the time it takes a simulator to mark a hit for a given coverpoint within its own data structure, before it can return and resume running the test itself. The test used to benchmark this takes two coverage points which are constructed using either SystemVerilog's coverage sublanguage or Graphcore's custom C++ coverage backend. Each coverpoint is divided into 256 bins. 16-bit signal values generated using SystemVerilog's *$urandom* can then be sampled by these coverpoints, with the value of the top 8-bits being sampled by the first coverpoint and the lower 8-bits by the second coverpoint. The same seed and therefore identical values are sampled for both methods. The time taken to mark a hit for the cross coverage of these two coverpoints is then measured across all simulators.

3) String Formatting & Messaging

The cost of formatting strings is also investigated, contrasting the performance of SystemVerilog's *$sformatf* and the *fmt* formatting library in C++ [4]. The time taken for 1,000 strings to be consecutively formatted will be recorded as one measurement, then averaged to find a time cost per string. This will help account for noise in each measurement by minimizing the number of DPI calls required to start and stop measurements. The variation in the time taken for a DPI call is significant enough that it could distort the average time taken to format a string using *$sformatf*. Message statements with a verbosity below the global logging level are not formatted when lazy logging is used, as they will never be printed. The amount of time which could be saved through the use of lazy logging is also considered [5].

The cost of messaging using SystemVerilog's *$display* and C++'s *printf* is evaluated. This can be determined by measuring the time taken for each messaging implementation to print the same set of strings.

III. Results

A. Cost of DPI invocations

In order to ensure the timing measurements for C++ and SystemVerilog tests were comparable, the time cost of the DPI calls when starting and stopping measurements must be quantified. Figure 3 shows measurements of the time taken for *"start_measurement"* and "*stop_measurement*" to be called sequentially, with each data point displaying the average of 100 of these empty timing measurements. Two of the commonly used EDA tools were used to record these measurements. The average time cost of the measurement infrastructure over time can be seen in Table 1.

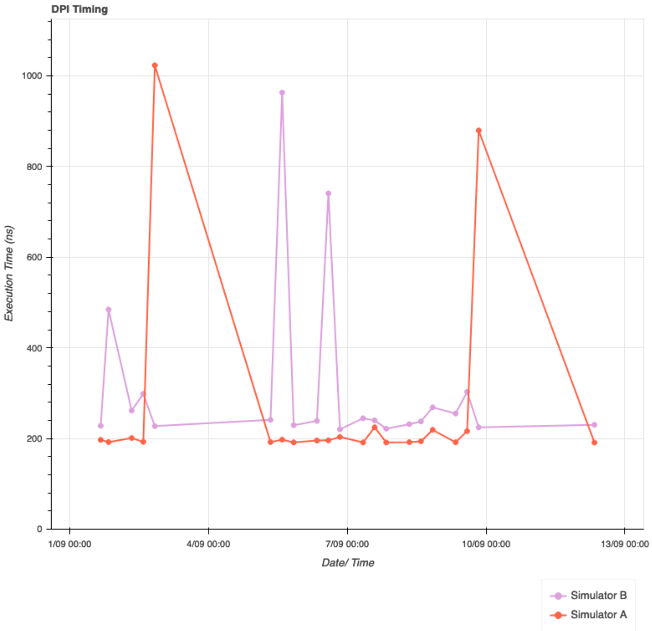

Figure 3 Data displaying the time taken (ns) for a DPI call to be performed over time. Measurements are taken for two different simulator tools.

	Simulator A	**Simulator B**
Mean (ns)	270.111	313.846

Table 1 The mean value (ns) of the time taken to perform a DPI call, with data taken from Figure 3 for each of the simulators.

B. Transaction Layer Modelling

The same two simulators were used to benchmark the time penalty incurred when transporting a 32-bit packet through a TLM network. It is noticeable in Figure 4 that the packet was transported through the C++ implementation of the TLM network much faster than for the SystemVerilog equivalent. The data in Table 2 shows a decrease in the time penalty incurred by factor of almost 5 for Simulator A and just over 21 for Simulator B. The scale of this difference is large enough that the 400 ns cost of the additional DPI calls used to start and stop the SystemVerilog model's timing measurements only has a marginal effect.

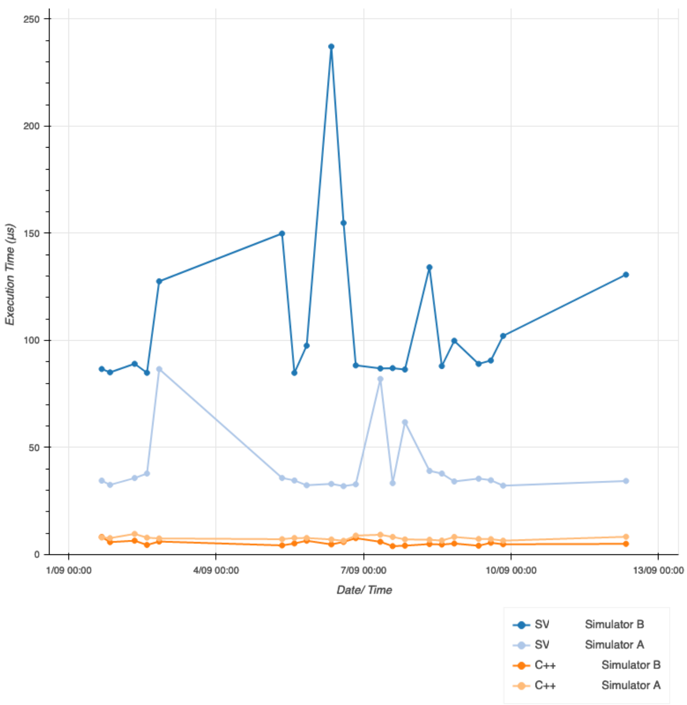

Figure 4 Data displaying the time taken (µs) for a 32-bit packet to be transported through 100 ports and 100 exports connected sequentially. This is repeated for two different simulators, as well as models implemented in SV and UVM, or C++.

SystemVerilog	**Simulator A**	**Simulator B**
Mean (µs)	40.615	108.597
C++		
Mean (µs)	7.707	5.436

Table 2 The mean value (µs) of each set of data forming a trace in Figure 4.

C. Functional Coverage Collection

Collecting the cross coverage described in Section II.B.2) resulted in CPU cache hit rate of almost 100%. This is due to the limited size of the testbench used to microbenchmark this action. A typical verification testbench would normally use far more memory and consequently lower cache hit rate. In order to make the benchmark more representative of coverage collection within a realistic simulation, a large unused array was instantiated each time data was sampled. This revealed that Simulator A had a significantly lower cache hit rate than Simulator B when collecting coverage in SystemVerilog. Figure 5 shows that Simulator A is drastically slower to record a cross coverage hit than Simulator B. Table 3 shows that this difference is of multiple orders of magnitude.

Large unused arrays were also instantiated for the coverage collection test when replicated in C++. Figure 5 shows that this was faster than the SystemVerilog equivalent, regardless of the simulator tool used – each took an average time of around 2.5 µs to record a cross coverage hit.

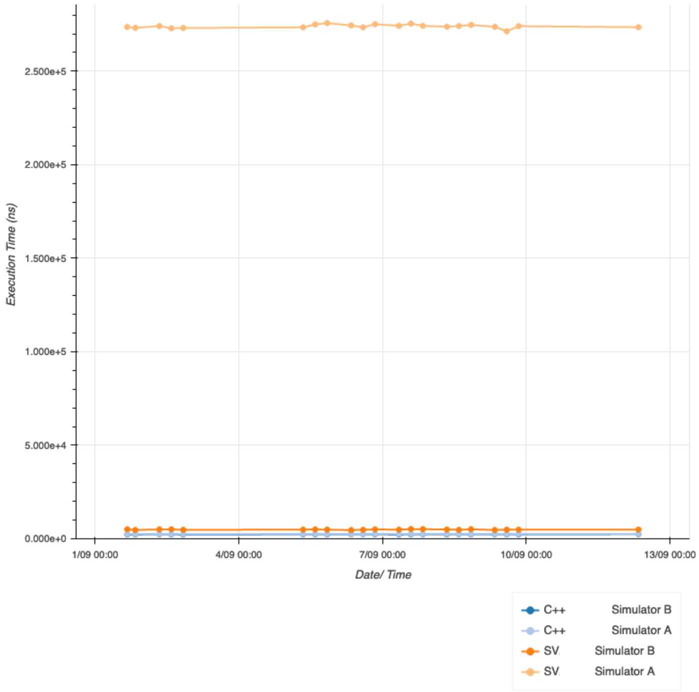

Figure 5 Data displaying the time taken (ns) for the cross coverage of two 8-bit coverpoints to be recorded. This is done for coverpoints instantiated in either SV or C++, with each being measured with two different simulators.

SystemVerilog	**Simulator A**	**Simulator B**
Mean (µs)	273.95	492.28
C++		
Mean (µs)	2.52	2.37

Table 3 The mean value (µs) of each set of data forming a trace in Figure 5.

D. *String formatting*

Figure 6 and Table 4 display various measurements for the time taken to format a single string, using either the C++ *fmt* library or SystemVerilog's *$sformatf*. Formatting is clearly faster in C++ than SystemVerilog, no matter the simulator tool used.

As a further optimisation, one could choose to perform formatting only if the message is going to be printed i.e. it matches the specified logging verbosity and filters. This functionality is commonly referred to as lazy messaging. Even if the $sformatf system task is variadic, SystemVerilog as a programming language does not support variadic arguments. This deficiency prevents the implementation of a lazy formatting feature in UVM whilst doing so in C++ or other programming languages would be trivial. The lost potential saving here is given by the total formatting time in SV in each simulator (374 ns and 976 ns respectively). This lost time could quickly accumulate in testbenches with significant debug messaging implemented.

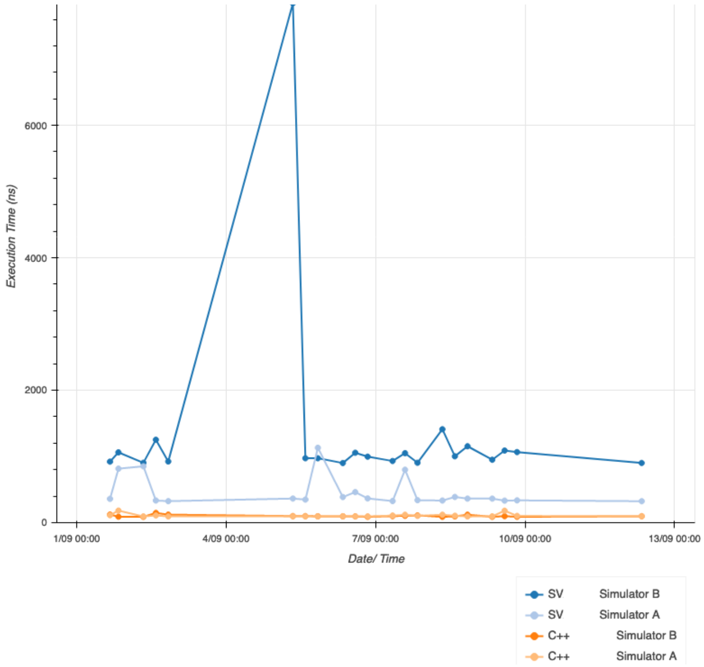

Figure 6 Data displaying the time taken (ns) for a string to be formatted using either the C++ fmt library or SystemVerilog's *$sformatf*.

SystemVerilog	Simulator A	Simulator B
Mean (ns)	459.018	1345.363
C++		
Mean (ns)	108.077	100.961

Table 4 The mean value (ns) of each set of data forming a trace in Figure 6.

E. String printing

Figure 7 shows the time taken for a simple message to be printed using either the classic C *printf* or *$display in SV*. Table 5 indicates that printing in C++ is approximately 2 times faster than in SystemVerilog.

The cost of messaging might not appear to be the most obvious limiting factor when considering the performance of verification simulations. However, the timing measurements displayed above show that the time taken to print a message is of the same order of magnitude as both coverage collection and TLM. Simulation logs can contain thousands of messages, so the messaging speedup offered by C++ could significantly reduce the amount of time spent logging.

In fact, even DPI calls with a string to be passed to *printf* significantly outperform *$display* for both vendors evaluated. This measurement indicates an obvious low hanging fruit that EDA vendors could look to rectify.

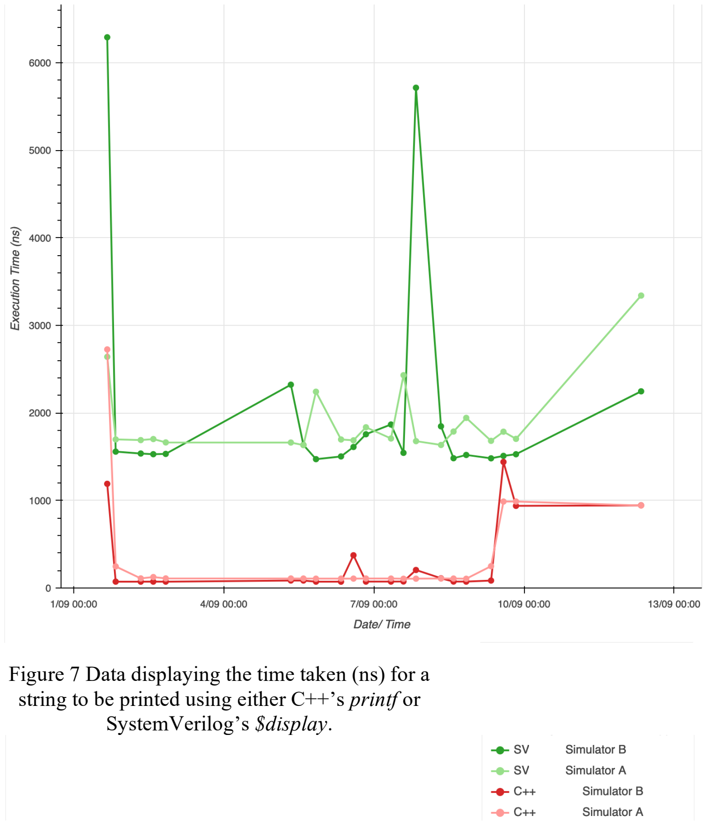

Figure 7 Data displaying the time taken (ns) for a string to be printed using either C++'s *printf* or SystemVerilog's *$display*.

SystemVerilog	Simulator A	Simulator B
Mean (ns)	1897.467	2071.053
C++		
Mean (ns)	370.424	298.526

Table 5 The mean value (ns) of each set of data forming a trace in Figure 7.

IV. Conclusion

For every action which has been investigated in this paper, the C++ implementations have been consistently faster. This is true for both simulator tools used. Additionally, for each of the benchmarked actions there is much more variance in the measurements taken for the SystemVerilog and UVM tests compared to their corresponding C++ counterparts. This is visible by looking at each of the figures in this section, and conclusively shown by the data in each of the tables. This is true regardless of the simulator tool used.

The results above highlight the simulation time which could be saved when each of these actions are performed once using C++, rather than SystemVerilog and UVM. The cumulative time saved through utilizing faster implementations of each commonly performed action could therefore be substantial, potentially saving many compute hours.

The cost of compute along with engineering time spent waiting on an interactive run is ever increasing due to ever more complex designs. The industry as a whole could do well to pay more attention to compute cost of the commonly accepted flows and methods.

References

[1] M. Cieplucha, "Metric-driven verification methodology with regression management," *Journal of Electronic Testing Volume 35,* p. 101-110, 2019.

[2] C. H. Stuart Hodgson, "CocoTb Documentation," [Online]. Available: https://docs.cocotb.org/en/latest/. [Accessed 6 June 2022].

[3] C. c. 1. Docs. [Online]. Available: https://en.cppreference.com/w/cpp/chrono/ [Accessed 6 June 2022].

[4] V. Zverovich, "Making string formatting fast," [Online]. Available: https://www.zverovich.net/2012/12/15/making-string-formatting-fast," [Online]. [Accessed 6 June 2022].

[5] Przemodev, "Lazy logging – how to properly log in Python?", 22 November 2021. [Online]. Available: https://www.przemodev.com/blog/articles/lazy-logging-how-to-properly-log-in-python. [Accessed 6 June 2022]