SEMARAK ILMU PUBLISHING

Journal of Advanced Research in Applied Sciences and Engineering Technology

Journal homepage:
https://semarakilmu.com.my/journals/index.php/applied_sciences_eng_tech/index
ISSN: 2462-1943

Python-Based DRAM Memory Controller Testbench: Pyuvm an Early Report

Dina Abdelbaky[1,*], Mohamed Dessouky[2], Ashraf Salem[1]

[1] Department of Computer and Systems Engineering, Ain Shams, University, Elsarayat, Abbaseya, 11517, Cairo, Egypt
[2] Department of Communication and electronics, Ain Shams, University, Elsarayat, Abbaseya, 11517, Cairo, Egypt

ARTICLE INFO

Article history:
Received 20 December 2023
Received in revised form 28 July 2024
Accepted 1 September 2024
Available online 20 September 2024

Keywords:

Hardware verification; SystemVerilog; UVM; Python; Cocotb; Pyuvm; DRAM; Memory controller

ABSTRACT

With Hardware Verification being regarded as a time-consuming complex task. Moreover, new design applications Such as machine learning are posing new challenges to hardware verification engineers. Adding on top of the above challenges the fact that the Universal Verification Methodology (UVM) has a class library that can be considered bloated. Python was suggested as an alternative to SystemVerilog, which is currently the main Hardware Verification language (HVL), used for building modern testbenches. Firstly, Cocotb was introduced almost ten years ago, then Pyuvm under two years ago. Python verification initiatives promise it is easier to learn(than traditional SystemVerilog), faster, and more productive to write. In this paper we explore the process of writing Python-based testbench, through writing DDR3 SDRAM Controller testbench. The testbench is implemented using the Pyuvm methodology running on top of Cocotb that's used to interact with the Design under test (DUT). The motivation of this paper is to explore the capabilities of Python based verification, and how much of similarity or difference it does bear compared to traditional SystemVerilog and UVM.

1. Introduction

The majority of modern electronic devices are built using the System on Chip (SoC) design paradigm, where the goal is to create a system by integrating pre-designed hardware and software blocks, which are frequently referred to as design intellectual properties (IPs). Given the increasing complexity of SoC design, Hardware Verification is experiencing new problems, such as shorter time to market [1]. This explains why hardware functional verification is currently viewed as a time-consuming task, taking roughly 40-50% of project time, with Application Specific Integrated Circuit (ASIC) ICs suffering from 18% re-spin and Field Programmable Gate Array (FPGA) projects suffering from 40% serious bug escape [1,2]. Python is being proposed as the software language used for hardware functional verification in order to boost hardware verification productivity and lower the

* *Corresponding author.*
E-mail address: g19093426@eng.asu.edu.eg

https://doi.org/10.37934/araset.52.2.176188

entrance barrier to hardware verification activities [4,5]. The verification life cycle of a SoC is typically composed of the following activities [1]:

i. Verification Planning
ii. Architecture Verification
iii. Pre-Silicon Verification
iv. Emulation FPGA Prototyping
v. Post-Silicon Verification

This paper includes the following sections, Section 2 introduces the origins of SystemVerilog and UVM. Section 3 introduces Python verification(Cocotb and Pyuvm). Section 4 introduces a comparison between important features in SystemVerilog and Python. Section 5 introduces the DUT (LiteDRAM) [6], and the testbench. Section 6 is the conclusion.

2. Origins of SystemVerilog and UVM

In the late 90s Verilog was widely used in the industry as Hardware Description Language (HDL) for simulation and synthesis, but due to growing complexity in designs VHDL and Verilog have shown to be inadequate for verification of those complex designs, their lack of—or poor—support for high level data types, object oriented programming, assertions, functional coverage and declarative constraints has prompted the need for creation of specialized languages for each or all of these areas [7-10] commercial efforts including “OpenVera” and “e” surfaced trying to fill this need. Due to those efforts being commercial they were not widely adopted by companies. Accordingly, “OpenVera” was donated to Accellera, and SV was born [8,10-12]. A reader of the SystemVerilog standard might be compelled to consider it as a single language, however, SystemVerilog actually is composed of three orthogonal languages [2]:

i. SystemVerilog Object-Oriented language for functional verification
ii. SystemVerilog Assertions (SVA) language
iii. SystemVerilog Functional Coverage (FC) language

Worth mentioning that an HVL is not the same or equivalent to a verification methodology, A Verification Methodology helps minimize the time necessary to meet the verification requirements, defines standards that will enable the creation of inter-operable verification environments and components. Using inter-operable environments and components is essential in reducing the effort required to verify a complete product [13-15]. After various commercial trials to create verification methodologies including Verification Methodology Manual (VMM) and Open Verification Methodology (OVM), The UVM standard from Accellera, a SystemVerilog class-based library emerged as a first representation of an alignment on verification methodology across the industry, from the major EDA suppliers and their ecosystems to many leading-edge users [13-16]. To enable design reuse UVM uses Transaction-Level Modelling (TLM) APIs to facilitate transaction level communication between verification components written in SystemVerilog as well as between components written in other languages such as e and SystemC, also TLM enables for the use of verification IPs.

3. Cocotb and Pyuvm Introduction

A key concept for any modern verification methodology is the layered testbench [7,8]. Accordingly, writing a testbench is considered a software task [4,8] then it would make sense to write a testbench in a software dedicated language [4,5]. The reason why Python was suggested as the new HVL [4,5] Writing Python is a fast, Python is a productive programming language, with the capacity to interface with other languages like MATLAB and C/C++. Python has a massive ecosystem represented in a large library of existing code, and a significant number of engineers who happen to be familiar with Python [4,5]. As mentioned earlier this paper uses Cocotb and Pyuvm for building the testbench. Cocotb stands for COroutine based COsimulation TestBench. In a Cocotb based verification environment, only the DUT runs in the Simulator, while the testbench is written in Python and interacts with the simulator through a VPI. Please refer to Figure1 and Figure 2, where Figure1 shows a traditional testbench while Figure 2 explains the interaction between a Python based testbench interacting with DUT.

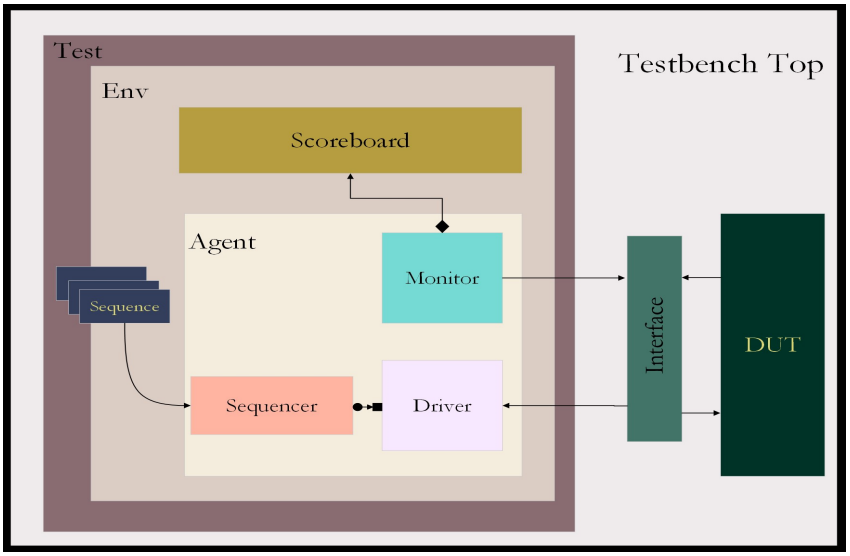

Fig. 1. UVM Testbench Architecture

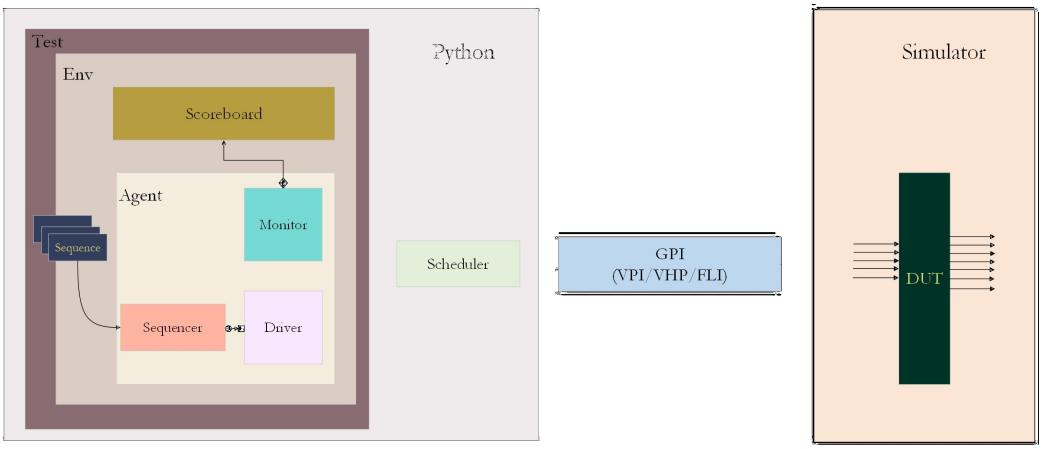

Fig. 2. Pyuvm Testbench Architecture

Pyuvm is a re-implementation of the UVM class library; this re-implementation uses Python instead of SystemVerilog. Pyuvm relies on Cocotb to gain handle to the DUT and to communicate with the simulator and schedule simulation events. The most frequently used UVM components are

implemented in Pyuvm by taking advantage of the fact that Python is not strongly typed and does not call for parameterized classes [4].

4. SystemVerilog vs. Python and Cocotb

In order to carry out first level assessment for the suitability of Python (or any other Software language) for hardware verification, we need to revisit a fact that we mentioned earlier in this paper: Verilog and VHDL were deemed inadequate for hardware verification due to lack of support of high-level data types, randomization and assertions. Secondly, we need to revisit the structure of the layered testbench mentioned in [8].The proposed architecture shows the need for OOP to allow for building the testbench components in a maintainable way, also the need for concurrency, and data exchange between testbench components. With this being said, we introduce a comparison between SystemVerilog [17] and Python(represented in Python language itself, combined with the added features and data types provided by the Cocotb package) [4,18] covering the above aspects.

4.1 Built-in Data Types

Software-like data types are important for building modern testbenches, yet hardware data types like: bit, and bit arrays are nice to have in an HVL; various protocols access data on bit basis.

Table 1
Built-in Data Types

SystemVerilog	Python and Cocotb
• SystemVerilog includes bit- addressable data types like: - reg, wire, and logic which happen to be 4-state unsigned. - bit which happens to be 2-state unsigned. • SystemVerilog includes: Software-like data types like: - integer which is signed 32 bit wide and happens to be 4 state. - byte which is signed 8 bit-wide and happens to be 2-state. - shortint which is signed 16 bit-wide and happens to be 2-state. - int which is signed 32 bit-wide and happens to be 2-state. - longint is signed 64 bit-wide and happens to be 2-state	• Python has built-in numeric types which are: integers, floating point numbers, complex numbers. In addition, Booleans are a subtype of integers. As we can Python does not have built-in Bit data types. However, there are more than one library independently developed for bit data types ex: Bit Vector. • Cocotb Introduces more data types which are Logic which is 4 state data type, Bit which is 2 state data type similar to SystemVerilog. • Cocotb also Supports Logic Array, which is equivalent to wire vector or a bit vector in SystemVerilog.

4.2 Aggregate Data Types

In the majority of hardware designs memories and FIFOs always exist, which creates a need to abstract those data structures properly in a testbench. Also, for the control path elements, a testbench needs to store expected transactions and compare them to received transactions, this also create a need for aggregate data types like arrays to store transactions in them.

Table 2
Aggregate data types

SystemVerilog	Python and Cocotb
• SystemVerilog offers various kinds of arrays [Data structure that all of its elements are homogeneous and of the same kind]: - fixed size arrays, packed and unpacked both are supported - Dynamic Arrays. - Associative Arrays. - Queues. • SystemVerilog supports packed and unpacked structs	• Python Supports lists which can be used to implement 2d and multi-dimensional arrays, list can contain elements of same or different data types. • Python support dequeues, even though from functionality perspective a list can act as a queue, dequeues act as an efficient way to implement a queue structure where can you pop elements in front and pop from back more effectively as a list. • Python Supports dictionaries which can be used to implement associative arrays. • Python has a module called arrays, that defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. • Python's library Numpy also supports arrays of fixed size and basic types • Cocotb also supports array type, arrays in Cocotb are of fixed size, can store any data types, elements of the array can be of the same type or of different types, arrays in Cocotb can't change size. • Python contains a struct module which converts between Python values and C structs represented

The list in Python covers the majority of the features of the following structures of SystemVerilog combined: Static Arrays, Dynamic Arrays Queues and some functionality of the struct.

4.3 Parallel Execution

Components in the testbench always need to be able to execute concurrently, for example a driver and monitor need to be operating simultaneously.

Table 3
Parallel execution

SystemVerilog	Python and Cocotb
• SystemVerilog provides multi-threading techniques through:	• Cocotb Provides us with:
- fork ,join: Finishes when all children threads are over	- start(),
- fork, join any: Finishes when any child thread gets over	- start soon(),
- fork, join none: Finishes soon after child threads are spawned	- combine() to enable concurrent execution.

4.4 Interprocess Synchronization and Communication

Components of the testbench always need to exchange data with each other, to synchronize operation and to pass data between different layers of the testbench, an example of that could be an exchange of a transaction between a monitor and a scoreboard. All of this data exchange and control synchronization is called interprocess communication (IPC).

Table 4
Interprocess synchronization and communication

SystemVerilog	Python and Cocotb
• SystemVerilog supports: - Events - Semaphores - Mailboxes Note: UVM uses TLM for data exchange between testbench components like sequencers and drivers	• Cocotb Supports: - Events - Locks - Queues (equivalent in behaviour to SV mailboxes) Note: Pyuvm uses TLM for data exchange between testbench components like sequencers and drivers

4.5 OOP Concepts and Language Basics

Table 5
OOP Concepts and Language Basics

SystemVerilog	Python and Cocotb
• SystemVerilog is compiled. • SystemVerilog move data bits between variables, accordingly they need to match in size. • SystemVerilog does not support multiple inheritance • SystemVerilog supports parameterized classes, which uvm uses for example, sequencers drivers are both parameterized by the req and resp (seqitem they are expected to exchange)	• Python is interpreted. • Python moves object handles between variables; in Python everything is an object --A variable of integer value is an instance of Class(integer) . • Python is dynamically typed. • Python supports multiple inheritance. • Python does not support parameterized classes

4.6 Constrained Randomization

In modern complex designs, constrained randomization plays an important role in the verification process. Constrained randomization, allows the verification engineer to find bugs that harder to find, compared to bugs found using directed testing [7,8].

Table 6
Constrained Randomization

SystemVerilog	Python and Cocotb
• SystemVerilog randomization is a built-in feature, randomization includes both local variables of a module (scope randomization) and class properties. • SystemVerilog supports randomization of various data types including: Integer Fields, Enum Fileds, Fixed size arrays, Variable size arrays. • SysytemVerilog supports various types of constraints including: dist constraints, soft constraints, inside constraint, unique and foreach constraint.	• Randomization can be implemented in Python using the module ”Random” which supports integers, distributions, weighted distributions, and choices from a sequence. • Randomization can also be implemented using Cocotb coverage library which supports basic randomization, and constructs like randomize with(), rand mode(), and allows for overriding of the methods pre randomize(), and post randomize().

4.7 Assertions

An assertion is simply a check against the specification of your design that you want to make sure never violates [20].

Table 7
Assertions

SystemVerilog	Python and Cocotb
• SysytemVerilog Provides various types of assertions that happen to be one in language yet used differently in the verification life cycle [2]: • assert for design check and for formal verification • assume for design constraint for formal verification • restrict for design constraint for formal verification SytemVerilog assertions are immediate, concurrent and deferred immediate assertions. SystemVerilog assertion can also be added by the RTL designer in the DUT to ensure correct operations in the DUT.	• Cocotb coverage provides assertion as part of code coverage checks. • Cocotb coverage assertion can assert a property or sequence, they behave like assert in SytemVerilog.

5. LiteDram the DUT

LiteDRAM is a lightweight DRAM Controller written in Migen, and implemented and verified in various FPGA projects [6]. LiteDRAM supports various DRAM devices including DDR3, DDR4, RPC, in this paper we used LiteDRAM core targeting DDR3 [21] devices. Lite DRAM is composed of front-end, Core, and PHY. For the front-end the core supports multiple interfaces, including AXI4, Wishbone, Native, and DMA. The Front-end can also include ECC port. The core is fully pipe-lined and supports multiple bank-machines, the core also issues periodic refresh and manages command scheduling accordingly.

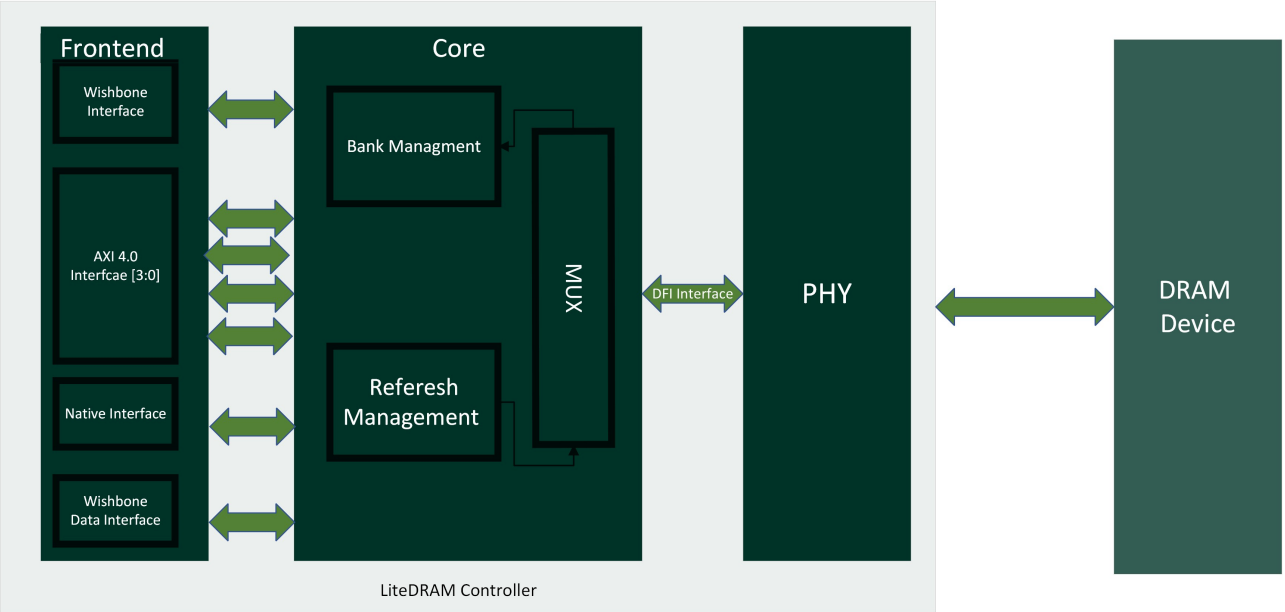

Fig. 3. LiteDram the DUT

6. Testbench Architecture

In this section we introduce Pyuvm to the reader and even offer the reader a comparison between traditional UVM and Pyuvm, we use our developed DDR3 Controller testbench to perform that introduction and comparison.

6.1 Connection to the DUT

Connecting the testbench to the DUT is achieved through setting the “TOPLEVEL” and “MODULE” Variables in the Cocotb Makefile.

Code Listing 1: Cocotb Makefile important variables

```
SIM ?= questa
TOPLEVEL_LANG ?= verilog
VERILOG_SOURCES=$(CWD)/../verilog/litedram_core.v
TOPLEVEL_LANG=$(TOPLEVEL_LANG)
MODULE := testbench #The file or the director that contains the test cases.
TOPLEVEL = litedram_core#The RTL top level module
```

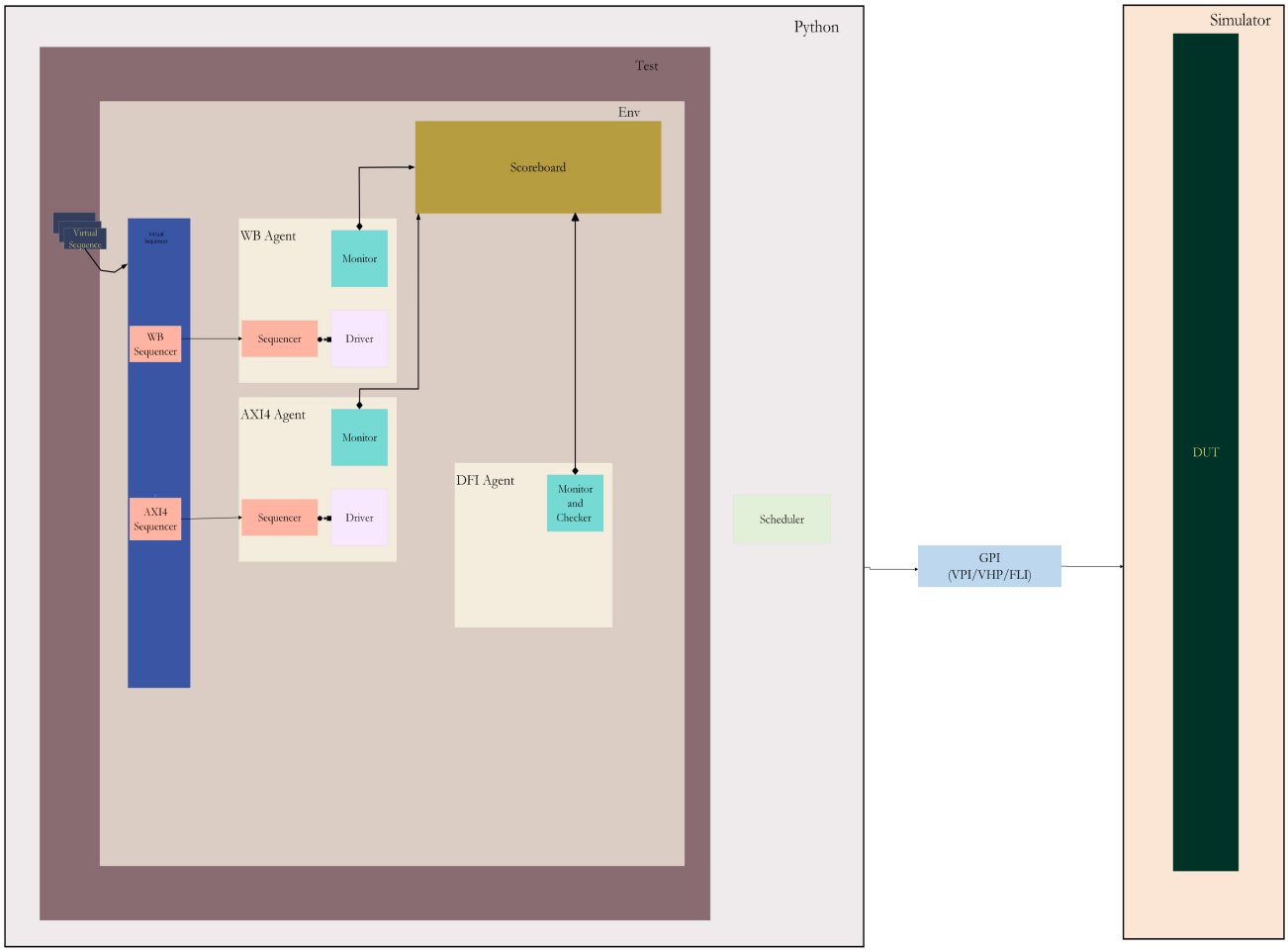

Fig. 4. Connection to the DUT

6.2 Pyuvm Tests

For each test case we use the decorator “@Pyuvm.test()” and we inherit from the class “uvm test”. The decorator “@Pyuvm.test()” allows Cocotb to discover the test case on its own and run it, without the need for the testbench developer to create or specify a regression script. please refer to code snippet below as it shows a simple example of running Pyuvm test case, the famous ”Hello, World” example.

Code Listing 2: Hello World example in Pyuvm

```
import Cocotb
from Cocotb import *
import Pyuvm
from Pyuvm import *
#add the "@Pyuvm decartor to make the test case discoverable"
@Pyuvm.test()
class DDRCtrlTest(uvm_test):
def build_phase(self):
ddr_ctrl_env = Env.create("ddr_ctrl_env",self)
ddr_ctrl_env_cfg = DDRCtrlEnvCfg.create("ddr_ctrl_env_cfg",self)
ConfigDB().set(self, "ddr_ctrl_env","DDRCtrlEnvCfg",ddr_ctrl_env_cfg)
async def run_phase(self):
self.raise_objection()
self.logger.info ("HelloWorld!")
self.drop_objection()
```

Table 8
Pyuvm tests

Traditional UVM	Pyuvm
• Test writer connects the DUT to the testbench through a virtual interface instance passed to the DUT (environment) by using the UVM configuration database. • The test writer imports uvm pkg then imports all classes from UVM library through using ::*. • traditional UVM relies on “run test();” accompanied by passing the test name to the command line through “+UVM TESTNAME=…..”. • The test writer uses “uvm component utils” macro to register a component within the factory. • Traditional UVM uses underscore naming style The ConfigDB() is parameterized. • UVM phases have been regarded as adding complexity [22] to UVM testbench writing process. • The “uvm factory.print()” an argument which is an integer value between 0, 1, 2 for type and instances override 0, for all user defined types plus types and instances overrides 1, for UVM types starting with uvm * are printed plus all of the previous mentioned types.	• Pyuvm relies on Cocotb to gain handle to the DUT. Cocotb ”Bus” can be used to group the signals neatly, the bus name can be passed to the driver and monitor through the configuration database. • The test writer imports uvm pkg, and Cocotb package and imports all classes from both packages through using ::*. • Pyuvm we use the “@Pyuvm.test()” decorator to mark a test case and make it discover-able to Cocotb to run it. Cocotb discovers all of the test cases in the testbench directory and runs them all. • Components utility macros are not supported. • Pyuvm uses underscore naming style, for the sake of compatibility. The ConfigDB() is not parameterized. Python doesn’t support parameterized classes. • Pyuvm phasing support is a simplification for the phases implemented in UVM. Currently, Pyuvm only supports: ”uvm build phase”, “uvm connect phase”, “uvm end of elaboration phase”, “uvm start of simulation phase”, “uvm run phase” (the only time-consuming phase), “uvm extract phase”, ”uvm check phase”, “uvm report phase”, “uvm final phase”. • Pyuvm supports the same factory behaviour as traditional UVM, additionally, Pyuvm supports the conversion of the logging of factory state by creating a string “str(uvm factory.print())”

From the above Code listings, we would like to highlight some difference between UVM and Pyuvm.

6.3 Environment and Sub-Components

The environment (env) is the top-level component of the verification component. The env contains agents, Wish- bone agent, AXI4 agent[3:0], all of them are active agents, and one more passive agent which is the DFI agent, as it only listens to the DFI Bus and does not need to drive any transactions to the memory:

i. Scoreboard:
The Scoreboard keeps track of predicted transactions and received transactions, while allowing out of order comparison.
ii. Drivers and Monitors:
The Drivers were created by extending the class “uvm driver” and Monitors were implemented by extending the (uvm component) class. Please refer to Code Listing3: showing Pyuvm driver.
iii. The Cocotb Bus:

Analogous to SystemVerilog’s Interface construct which is used to group signal together, Cocotb provides an extension library called “Cocotb bus” The class Bus allows a testbench developer to group signals of the same protocol together like SystemVerilog Interfaces, the BUS class accepts optional signals, allowing for future scalability as it contains a method for adding new signals to an existing Bus. An instance of Cocotb Bus was created for every protocol that the design supports, and passed to the corresponding driver and monitor. For the AXI4 Cocotb already provides an AXI4 Bus driver, which contains an API that drives write transactions to the AXI4 in the design, drive read transactions and collect read data.

Code Listing 3: Wishbone Driver implemented in Pyuvm

```
class WBDriver(uvm_driver):
def build_phase(self):
self.wb_bfm = WBfm("wb_bfm", self )
async def run_phase(self):
while True:
cmd = await self.seq_item_port.get_next_item() """start of Sequencer driver handshake"""
logging.info("WB Driver Transcation got from WB Sequencer")
logging.info(str(cmd))
 await ClockCycles(self.wb_bfm.dut.clk,cmd.wb_delay, rising=True)
await FallingEdge(self.wb_bfm.dut.clk)
self.wb_bfm.wishbone_bus.adr.value = cmd.wb_addr#Driving the Wishbone Address
"""Rest of Wishbone Transaction driving logic"""
logging.info("Driver Returned From WBFM Transcation driving")
 self.seq_item_port.item_done()#End of Sequencer Driver Handshake""
```

The Env components and its components are reporting classes, extended from ”uvm report object” accordingly we want to introduce the audience to reporting in Pyuvm.

Table 9
Environment and sub-components

Traditional UVM	Pyuvm
• UVM relies on reporting macro to report messages, ex 'uvm error(string ID, string MSG). • UVM has six verbosity levels: UVM NONE, UVM LOW, UVM MEDIUM (Default), UVM HIGH, UVM FULL, UVM DEBUG.	• Pyuvm relies on Python logging mechanism, the "uvm report object" creates an object named "self.logger", ex: self.logger.error("string MSG") • Python logging module has six levels: CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET. In order to change the logging level of any component "self.set logging level()" • Furthermore, Pyuvm enables the user to change logging format, logging handler. And sets an extra level for "uvm tlm analysis fifo" which is 5.

6.4 Sequences and Sequence Items

Sequence Items known as transactions are a crucial part of any testbench, as they contain data fields required for generating the stimulus, and for creating sequences by randomizing sequence items. Worth mentioning that even though uvm and Pyuvm are almost the same just re interpretation of each other in different languages, the sequence item class object highlights some interesting features in both languages:

i. Pyuvm does not support Field macros and Object utility macros, which should have been expected by the reader since Component utility macros are not supported.
ii. Pyuvm relies on the fact that python by default support functions like str, and eq to return a string containing the class member values, or to compare an instance of a class object to another, and accordingly Pyuvm authors encourages the reader to implement the mentioned methods, accordingly the "do print" method is not implemented in the "uvm object" class which means it cannot be overridden by the user.
iii. Pyuvm runs in the software side, not the simulator. Accordingly, "record()" and "do record()" methods are not implemented and cannot be overridden.
iv. Pyuvm authors do not implement some of the methods related to data formatting like "do pack()", and "do unpack()" as Python has libraries for doing this job in a more elegant way.
v. Pyuvm is built on Python which means the randomization options in Python are different than that of SystemVerilog, accordingly it was a personal choice to do multiple inheritance from both "uvm sequence item" and Cocotb coverage "crv.Randomized" to be able to implement randomization and constraints. The reader is advised to refer to [4] and see how the author implements randomization and use the sequence to constraint the transaction's sequence item fields. We found the style presented here to be easier than the way the author of [4] carried out this process.

Code Listing 4:sequence item implemented in Pyuvm

```
class AxiSeqItem(uvm_sequence_item,crv.Randomized):
def __init__(self, name):
super().__init__(name) crv.Randomized.__init__(self)
self.axi_addr = 0x0
self.axi_data=0x777766665555444433333222211110000
self.axi_burst = 0x0
self.axi_size = 0x4
```

```
    self.axi_len = 0x0
    self.axi_strb = 0xffff
    self.axi_id = None
    self.axi_wr_rd = 0x1
    self.axi_delay = 0x1
    self.add_rand("axi_strb", list(range(2**16)))
    self.add_rand("axi_addr",list(range(2**18)))
    '''Creating Data Kernal'''
    self.add_rand("axi_data", list( range( (2**16)-8) ) )
    self.add_rand("axi_wr_rd", list(range(2)))
    self.add_rand("axi_len", list(range(16)))
    self.add_rand("axi_size", list(range(5)))
    self.add_constraint(lambda axi_addr : (axi_addr)%16 == 0)
    def post_randomize (self):
    if (self.axi_len == 0x0):
    self.axi_burst = 0x0
    else :
    self.axi_burst = 0x1
    self.axi_data = self.axi_data\
    |(self.axi_data+1) <<16\
    ……..
    |(self.axi_data+7) <<112
    def __eq__(self, other):
    same = self.axi_data == other.data and self.axi_addr == other.addr and self.axi_strb ==
other.axi_strb
    return same
    def __str__(self):
    return f'{self.get_name()}: Addr: 0x{self.axi_addr:04x}  Data: 0x{self.axi_data:04x}  WR:
{self.axi_wr_rd}  Sel: 0x{self.axi_strb} Length : 0x{self.axi_len}  Burst : 0x{self.axi_burst}'
```

7. Conclusion

Through this paper we have shown that Python has less data structures than SystemVerilog, also Python is more forgiving than SystemVerilog, those two features could arguably make learning Hardware Verification using Python easier. Python and Cocotb Covers most of the language constructs of SystemVerilog, we also showed the reader than Pyuvm covers the minimum needed subset of the UVM class library to allow for design verification. Python has slightly different features than SystemVerilog like multiple inheritance, this might encourage verification engineers to revisit some of the existing verification practices. Python seems to be promising in the applications of verification that involve DSP applications, heavy Data Path applications, and machine learning, as Python contains more than one package that supports machine learning, which means the package would just be imported in the testbench and used directly. On the other hand, Cocotb and Python verification initiatives seems like a work in progress with current status which is the lack of strong randomization as in SystemVerilog and limitation on the support for SystemVerilog assertions binding. Also, there are very few python verification components and Python verification Ips [4], while there are plenty of UVC (Unified Verification components) available in UVM --commercial and

non-commercial-- and it has been shown before in [3] that hardware verification process consumes the highest amount of time in projects where UVCs are not available for reuse. Yet, the main advantage of Pyuvm and Cocotb is the fact that they are open-source this would provide the academic community, and fresh-grads the opportunity to learn hardware verification and develop more standardized reusable verification components.

Acknowledgement
This research was not funded by any grant.

References

[1] Chen, Wen, Sandip Ray, Jayanta Bhadra, Magdy Abadir, and Li-C. Wang. "Challenges and trends in modern SoC design verification." *IEEE Design & Test* 34, no. 5 (2017): 7-22. https://doi.org/10.1109/MDAT.2017.2735383
[2] Mehta, Ashok B. "ASIC/SoC functional design verification." *A Comprehensive Guide To Technologies and Methodologies* (2018). https://doi.org/10.1007/978-3-319-59418-7
[3] Foster, Harry. "2022 Wilson Research Group IC/ASIC functional verification trends." *White Paper. Siemens EDA,* (2022). https://blogs.sw.siemens.com/verificationhorizons/2022/10/24/part-2-the-2022-wilson-research-group-functional-verification-study/
[4] Salemi, Ray. "Python for RTL Verification: A Complete Course in Python Cocotb and Pyuvm." (2021).
[5] COroutine based COsimulation TestBench. "Welcome to Cocotb's Documentation! - Cocotb1.8.0 Documentation." (2023). https://docs.Cocotb.org/en/stable/
[6] Kermarrec, Florent, Sébastien Bourdeauducq, Jean-Christophe Le Lann, and Hannah Badier. "LiteX: an open-source SoC builder and library based on Migen Python DSL." *arXiv preprint arXiv:2005.02506* (2020).
[7] Bergeron, Janick. *Writing testbenches using SystemVerilog*. Springer Science & Business Media, 2007. https://doi.org/10.1007/0-387-31275-7
[8] Spear, Chris. *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media, 2008.
[9] Cummings, Clifford E. "SystemVerilog-Is This The Merging of Verilog & VHDL?." *www. sunburst-design. com/papers/CummingsSNUG2003Boston_SystemVerilog_VHDL. pdf* (2003).
[10] Flake, Peter. "Why SystemVeriog?." In *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, pp. 1-6. IEEE, 2013.
[11] Mehta, Ashok B. *Introduction to SystemVerilog*. Springer Nature, 2021. https://doi.org/10.1007/978-3-030-71319-5
[12] Rich, David I. "The evolution of SystemVerilog." *IEEE Design & test of computers* 20, no. 04 (2003): 82-84. https://doi.org/10.1109/MDT.2003.1214355
[13] Bromley, Jonathan. "If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language." In *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, pp. 1-7. IEEE, 2013.
[14] Meade, Kathleen A., and Sharon Rosenberg. *A practical guide to adopting the universal verification methodology (UVM)*. Cadence Design Systems, 2010.
[15] Salah, Khaled. "A UVM-based smart functional verification platform: Concepts, pros, cons, and opportunities." In *2014 9th International Design and Test symposium (IDT)*, pp. 94-99. IEEE, 2014. https://doi.org/10.1109/IDT.2014.7038594
[16] Accellera Systems Initiative. "Universal Verification Methodology (UVM) 1.2 User's Guide." *201411.*
[17] IEEE Standards Association. "IEEE standard for Verilog hardware description language (IEEE 1364-2005)." *http://standards. ieee. org/* (2006).
[18] Python. "The Python Tutorial," *Python 3.12.8 documentation,* (2023). https://docs.python.org/3/tutorial/index.html
[19] Fvutils. "GitHub - Fvutils/Pyvsc: Python Packages Providing a Library for Verification Stimulus and Coverage." *GitHub.* https://github.com/fvutils/pyvsc
[20] Mehta, Ashok B. *SystemVerilog Assertions and Functional Coverage*. Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-24737-9_23
[21] Jedec Solid State Technology Association. "DDR3 SDRAM (JESD 79-3E)." (2010).
[22] Sutherland, Stuart, and Tom Fitzpatrick. "UVM Rapid Adoption: A Practical Subset of UVM." *Proceedings of DVCon* (2015).