Towards Efficient Design Verification – Constrained Random Verification using PyUVM

Deepak Narayan Gadde
Infineon Technologies
Dresden, Germany
Deepak.Gadde@infineon.com

Suruchi Kumari
Infineon Technologies
Dresden, Germany
Suruchi.Kumari@infineon.com

Aman Kumar
Infineon Technologies
Dresden, Germany
Aman.Kumar@infineon.com

Abstract

***Abstract* —Python, as a multi-paradigm language known for its ease of integration with other languages, has gained significant attention among verification engineers recently. A Python-based verification environment capitalizes on open-source frameworks such as *PyUVM* providing Python-based UVM 1.2 implementation and *PyVSC* facilitating constrained randomization and functional coverage. These libraries play a pivotal role in expediting test development and hold promise for reducing setup costs. The goal of this paper is to evaluate the effectiveness of PyUVM verification testbenches across various design IPs, aiming for a comprehensive comparison of their features and performance metrics with the established SystemVerilog-UVM methodology.**

I. Introduction

With the continuous increase in complexity of System-on-Chip (SoC) designs, verification is becoming ever more challenging. As a result, the time required for verification experiences a significant upsurge. Additionally, there is a subsequent need to be more productive and efficient with limited manpower. Industry-utilized methodologies like Constrained Random Verification (CRV), Formal Verification (FV), and Metric Driven Verification (MDV) use SystemVerilog as a language construct. It provides numerous features like object-oriented programming and functional coverage, but learning the language has a steep curve especially for the freshers requiring good understanding of designs.

Fig. 1 shows SystemVerilog is the most complicated language in comparison with other programming languages, as it has 1315 specification pages and 248 keywords as per IEEE 1800-2012, with variations in the level of tool support across different Electronic Design Automation (EDA) vendors [1]. In summary, Universal Verification Methodology (UVM), which utilizes SystemVerilog, gets more complex in addition to its standard. On the other hand, Python is simple, less verbose, and easily integrable to Numpy, Pandas, and other open-source libraries. A recent study conducted by the Wilson research group [2] has shown that 23 % of all Application-Specific Integrated Circuit (ASIC) projects used Python for various project specific tasks.

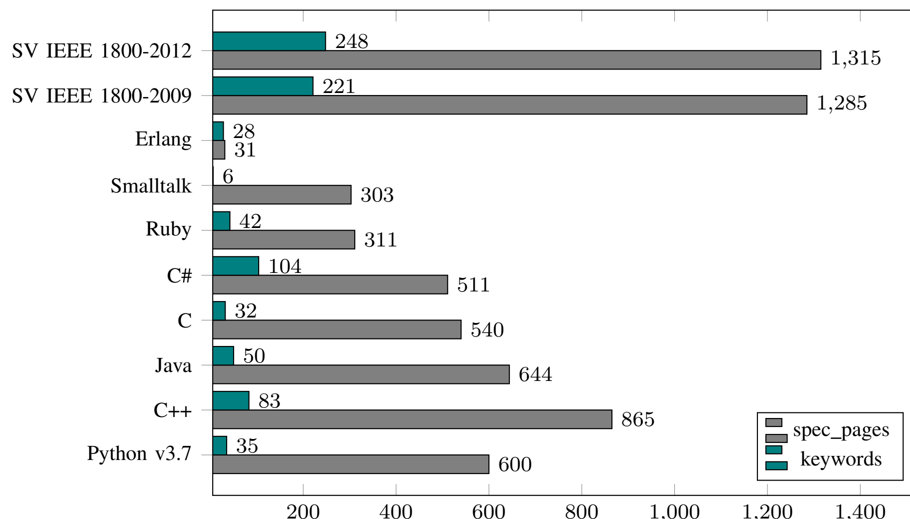


Fig. 1: Language complexity with respect to number of specification pages and keywords [1]

Like SystemVerilog-UVM, PyUVM implements UVM 1.2 IEEE specification in Python using its high level language features, and is built upon Cocotb to interface with the simulators. The main objective of this work is to assess the effectiveness of

Python testbench development using PyUVM and PyVSC libraries for the design Intellectual Properties (IPs), and to compare it with the existing state-of-the-art methodologies, such as UVM and CRV. The following are key highlights of the paper:

- Comparison of Systemverilog-UVM with PyUVM for testbench development of various design IPs
- Feasibility of PyVSC library for enhanced coverage declaration and CRV
- Comparison of performance metrics in terms of simulation run-time
- Empirical observations made during the development of Python testbenches

The structure of the paper is as follows: Section II summarizes various prior works related to python based testbenches. Section III details the design IPs used for the testbench development in both Systemverilog-UVM and PyUVM. Section IV explains the implementation of our approach in creating PyUVM testbenches with an example. Section V discusses the results and empirical observations made during our research. At the end, Section VI concludes our current work with future scope.

II. Related Work

Prior work [3] has created the PyUVM framework for a RISC-V Single Cycle Core and has encouraged the features supported by the methodology. The author in [4] discusses how the verification environment created in Python helps to reuse the test sequences across testbenches. In [5], the author proposed a flow using the *Verifog* tool to catch bugs at the earliest stages of the design phase without developing testbenches. *uvm-python* is a port of SystemVerilog-UVM 1.2 to Python and Cocotb tested with Icarus Verilog and Verilator [6]. Additionally, research [7] has compared Python with SystemVerilog to show its significance in design verification. The authors provide a detailed comparison between both Hardware Verification Languages (HVLs) in terms of design hierarchy, coverage constructs, and their performance during simulation. The feature comparison from the paper is given below in Table I.

TABLE I: Comparison between SystemVerilog and Python [7]

Feature	**SystemVerilog**	**Python**	**Remarks**
Declaration of data types	Static	Dynamic	Python allows undeclared variables and perform any operation on them. Additionally, it has advanced data structures e.g., tuple and dictionary, unlike SystemVerilog.
Supported types of logic	*0, 1, X, Z*	*X, Z, U, W*	Python-Coroutine based cosimulation testbench (Cocotb) needs BinaryValue object for these logics
Parameterization and size of the variable	Required	Not required	If size is not declared in SystemVerilog, data may be lost after an assignment to a different size than specified
Styles of control flow	*begin, end*	Proper indentation	*elif* in Python replaces case in SystemVerilog/Verilog
Functions	Not objects	Callable objects	Function in SystemVerilog are not objects and cannot be stored or passed directly as arguments
Exceptions	Not supported	Supported	In Python, exceptions are caught with *try/except/finally* blocks
Libraries	-	Large	Create reference model for any complex design easily
Interpreted	No	Yes	It allows to restart the simulator without recompiling and edit tests while it is running
Design Hierarchy	Includes top testbench	Does not include top testbench	It limits debugging capabilities, since tracing back signals in the testbench is not possible

The related works elaborated on above mostly discuss PyUVM implementation but none of them examines how it is different from commonly used methodology i.e., SystemVerilog-UVM in functional verification. In this work, we try to address this by comparing various features of PyUVM with SystemVerilog-UVM, and their performances are analyzed in terms of simulation run-time.

III. Designs

We have carefully chosen three distinct designs in our research for verification implementation considering several important aspects i.e., Arithmetic Logic Unit (ALU), Analog-to-Digital Converter (ADC), and Error Correction Code (ECC). These may aid in our decision about compatibility of PyUVM testbenches in comparison with SystemVerilog-UVM methodology. A short explanation for the design IPs is given below.

A. ALU

The 32-bit ALU is a hardware unit that is designed using Verilog and can perform various computations on 32-bit input data. These include two arithmetic operations: addition and subtraction; and six logical operations, including NOT, AND, OR, XOR, NAND, and NOR. This IP has 4 inputs, two of which are data buses a and b, plus a control bus op, and a clock signal. The output of the design module would be a data bus r. Fig. 2 displays the block diagram of the 32-bit ALU. Due to its computational capabilities, it is commonly used in modern processors.

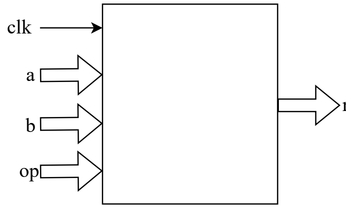


Fig. 2: ALU

B. ADC

The ADC module is primarily meant to convert analog input values to the corresponding digital data with 16 bit resolution. To improve the resolution and reduce the noise of the conversion, an oversampling feature is also added to the ADC. Control and Status Registers are available to configure the oversampling factor of 1, 2, 4, or 8, with 1 indicating no oversampling. The conversion of *analog_in* can be triggered via another register bit and the result sent out on the *digital_out* bus as shown in Fig. 3.

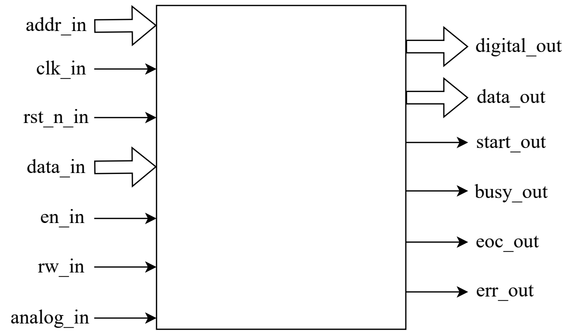


Fig. 3: ADC

TABLE II: Description of Register Model used in ADC design IP

Register	Address	Width	Access	Reset Value	Details
Dummy	'h0	8	RO	'h0	Dummy register - not used
Config	'h1	8	[7:2] RO	'h0	Reserved - not used
			[1:0] RW		Oversampling factor (1,2,4,8)
Trigger	'h2	8	[7:1] RO	'h0	Reserved - not used
			[0] RW		Start ADC conversion

The registers can be accessed via a simple register interface with address, data and read/write signals. The registers are 8 bits in width. To write to a register, *rw_in* must be set, *addr_in* should be driven by the target register address and *data_in* should hold the value of data that needs to be written to the register. To read from a register, *rw_in* must be de-asserted and the *addr_in* bus should be provided with the target register address. The *data_out* bus from the register interface will hold the value read from the target register. The register description is mentioned in Table II. Register address 0 is a dummy register and kept for future use. Register address 1 is the configuration register where bit 0 is used to trigger an ADC conversion, bits 1 to 2 are used to set the oversampling factor, and bits 3 to 7 are reserved bits.

C. ECC

ECCs are extensively used with the aim of protecting against soft-errors in automotive products that are crucial to safety. These errors arise either in the logic or in data due to radiation, electrical glitches, or electromagnetic interference, which can occur either during the production process or while the device is being used.

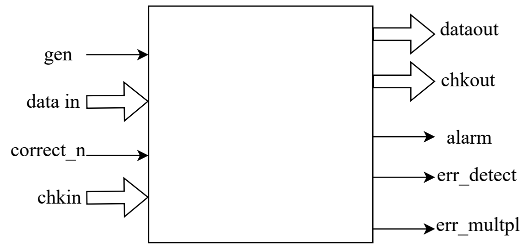


Fig. 4: ECC Core

The ECC design implements a Single-Error Correction, Double-Error Detection (SECDED) over a parameterized range of word widths. It involves incorporating additional bits, called check bits, into each storage word (such as in RAM). During the writing process, these encoded check bits are written alongside the data bits. Upon reading a word, the check bits are utilized to determine if there are any errors and, if there is a single-bit error, which specific bit contains the error. This particular ECC module is capable of detecting all one or two-bit errors (including check bits) and correcting all single-bit errors (including check bits). However, this ECC does not guarantee error detection if more than two bits in a word are faulty (including check bits). In such cases, the error syndrome may match the error syndrome of a single-bit error, causing the ECC to miscorrect a single-bit error that does not exist.

The block diagram of the ECC core module is shown in Fig. 4. The same module can be configured as an encoder or a decoder based on the input signal *gen*. When the *gen* signal is set to 1, it configures the design as an encoder otherwise as a decoder. The bus *datain* is the data word to check during the *check mode*, or data from which check bits are generated during *generate mode*. The primary output of the module is *dataout* which can be corrected if an error is detected and *correct_n* is asserted. Other outputs include error check flags i.e., *err_detect* and *err_multpl* if the ECC core module is used as decoder.

IV. Verification Implementation

With a clear understanding of the design Intellectual Propertys (IPs), we proceed with the verification implementation phase. In PyUVM as shown in Fig. 5, the testbench software is the PyUVM testbench, the proxy is implemented in Cocotb, where Cocotb connects Python to simulators through Verilog Procedural Interface (VPI) and VHDL Procedural Interface (VHPI). Hence both Python and the Design Under Test (DUT) which runs in the simulator share the same proxy interface, the transactions from the testbench are called using the Python coroutines (driver and monitors Bus Functional Models (BFMs)) which also ensures DUT is not busy.

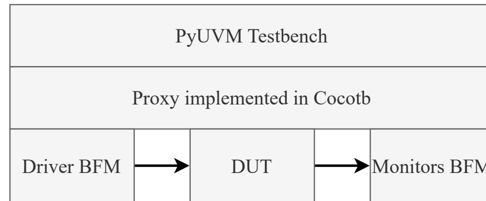


Fig. 5: Proxy-driven PyUVM testbench [8]

Figure 6 illustrates the PyUVM testbench architecture used in our work, which depicts the connections and communication between UVM components. The following subsections present a concise explanation of the PyUVM testbench implementation for the ECC design IP.

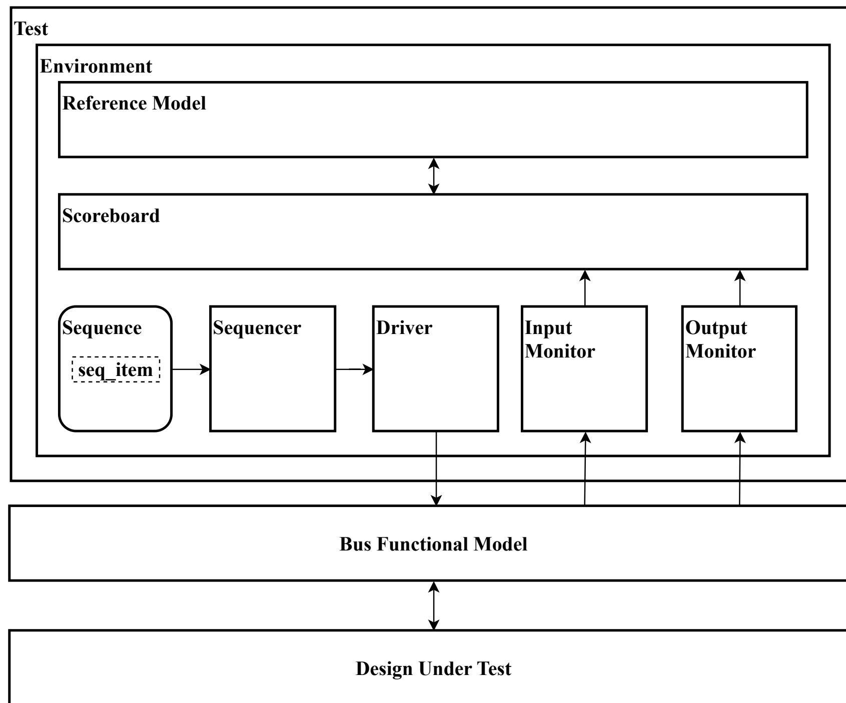


Fig. 6: Testbench Architecture of PyUVM using BFM class

A. ECC BFM

To enable access to the DUT and to abstract the Universal Verification Methodology (UVM) testbench code, a BFM class has been created. This BFM has been implemented using coroutines in Cocotb. Within the *__init__()* function of the BFMs class, a handle to the top module of the design is included. Additionally, a clock with a parameterized time period is generated as further explained in Listing 1, line 2-3. To facilitate the transfer of transactions between the testbench and the DUT, three queues have been declared, as specified in Listing 1, line 4-6.

Listing 1: __init__() function of ECC BFM class

```
1  def __init__(self):
2    self.dut = cocotb.top
3    cocotb.start_soon(Clock(self.dut.clk, CLKPERIOD, units="ns").start())
4    self.driver_queue = Queue(maxsize=1)
5    self.inp_mon_queue = Queue(maxsize=0) # Infinitely long
6    self.out_mon_queue = Queue(maxsize=0) # Infinitely long
```

As discussed before, the details of important coroutines inside the BFM-based class are as below.

1) initialization(): The purpose of this function is to initialize the signals of the DUT.

2) driver_bfm(): The implementation of the driver BFM which is included in the BFM class, is demonstrated in Listing 2. By utilizing the *RisingEdge* and *FallingEdge* triggers, this coroutine can function as an RTL BFM. The BFM is looped upon a clock positive edge. Within the *try* block, a timed *get()* operation is executed on a *Queue* if a transaction can be found within it, whereas the loop will continue to the next iteration and wait for the next positive clock if there are no transactions in the *Queue*. As shown in line 6, this transaction is assigned to the Encoder input signal *datain*. In line 7 of Listing 2, the *gen_random_index(data_in_gen)* method is called which inserts bit flip(s) randomly in either data or check bits after encoding.

Listing 2: Driver coroutine in ECC BFM class

```
1  async def driver_bfm(self):
2    while True:
3      await RisingEdge(self.dut.clk)
4      try:
5        self.data_in_gen = self.driver_queue.get_nowait()
6        self.dut.enc_datain.value = self.data_in_gen
7        self.gen_random_index(self.data_in_gen)
8      except QueueEmpty:
9        continue
```

3) inp_mon_bfm(), out_mon_bfm(): Two monitors are created utilizing coroutines from Cocotb in order to populate two infinite, long queues with input and output signals originating from the DUT. The implementation details of these monitors can be found in Listing 3 and Listing 4, respectively. These monitors are looped at the negative edge of the clock. The presence of queues permits the UVM threads to make blocking calls into these asynchronous coroutines.

Listing 3: Input Monitor coroutine in ECC BFM class

```
1 async def inp_mon_bfm(self):
2   while True:
3     await FallingEdge(self.dut.clk)
4     inp_tuple = (get_int(self.dut.enc_datain),get_int(self.dut.enc_chkin))
5     self.inp_mon_queue.put_nowait(inp_tuple)
```

Listing 4: Output Monitor coroutine in ECC BFM class

```
1 async def out_mon_bfm(self):
2   while True:
3     await FallingEdge(self.dut.clk)
4     out_tuple = (get_int(self.dut.dec_err_detect), get_int(self.dut.dec_err_multpl),
          get_int(self.dut.dec_dataout), get_int(self.dut.dec_chkout))
5     self.out_mon_queue.put_nowait(out_tuple)
```

4) start_bfm(): This method, which is also a part of the BFM class, is depicted in Listing 5 initiates the simultaneous execution of all coroutines during the run phase. It is launched in the build phase of the test, which is defined at the top of the testbench, as shown in Listing 6, line 6.

Listing 5: Method to start all coroutines

```
1 def start_bfm(self):
2   cocotb.start_soon(self.driver_bfm())
3   cocotb.start_soon(self.inp_mon_bfm())
4   cocotb.start_soon(self.out_mon_bfm())
```

B. Test

Figure 6 depicts that the test comprises all the necessary UVM components essential for the testbench development. It is designated with the decorator[1], i.e., *@pyuvm.test()*, to recognize itself as a test during the compilation process. The implementation of the test is shown in Listing 6, which extends the *uvm_test*.

During the build phase, the ECC BFM class object and *EccEnv* are created, and the PyUVM version of ConfigDB, which is also a part of the *uvm_component*, is set.

The run phase in all UVM components is asynchronous and is declared as a coroutine[2] function using *async def*. As it requires time, *self.raise_objection()* and *self.drop_objection()* are used to inform Cocotb of the simulation status (Started/Finished). Between these methods, a sequence is created and transferred to the sequencer to initiate, as shown in lines 12-14, Listing 6.

Listing 6: Test top of PyUVM testbench

```
1  @pyuvm.test()
2  class BaseTestEcc(uvm_test):
3    def build_phase(self):
4      bfm = EccBfm()
5      ConfigDB().set(None, "*", "BFM", bfm)
6      bfm.start_bfm()
7      self.env = EccEnv.create("env", self)
8    async def run_phase(self):
9      self.raise_objection()
10     seqr = ConfigDB().get(self, "", "SEQR")
11     bfm = ConfigDB().get(self, "", "BFM")
12     seq = EccSeq("seq")
13     await seq.start(seqr)
14     await ClockCycles(bfm.dut.clk, 3)
15     self.drop_objection()
```

C. Environment

The *EccEnv* class is an extension of the *uvm_env* class, which encapsulates the necessary components required for the testbench, as outlined in Listing 7. Within the build phase (Lines 3-8), an instance of the *uvm_sequencer* is created to queue

[1]Decorators use @ followed by the decorator name before a function or class declaration. When invoking the decorated function or class, the decorator runs first, and its output substitutes the original.

[2]A coroutine function is a Python function declared with async def.

and pass sequence items to the driver, and the Driver, Monitors, and Scoreboard are instantiated. In the Monitors, the name of the proxy functions that get the data they monitor is passed.

During the connect phase, the exports are connected to the ports, as exemplified in lines 10-12, Listing 7. The build phase is a top-down phase, while the connect phase is a bottom-up phase.

Listing 7: Environment of PyUVM testbench

```
1  class EccEnv(uvm_env):
2    def build_phase(self):
3      self.seqr = uvm_sequencer("seqr", self)
4      ConfigDB().set(None, "*", "SEQR", self.seqr)
5      self.driver = Driver("driver", self)
6      self.inp_mon = InputMonitor("inp_mon", self, "get_inp")
7      self.out_mon = OutputMonitor("out_mon", self, "get_out")
8      self.scoreboard = Scoreboard("scoreboard", self)
9    def connect_phase(self):
10     self.driver.seq_item_port.connect(self.seqr.seq_item_export)
11     self.inp_mon.ap.connect(self.scoreboard.inp_export)
12     self.out_mon.ap.connect(self.scoreboard.out_export)
```

D. Monitor

Figure 6 demonstrates that the PyUVM testbench implementation may include two monitors, namely the *Input Monitor* and *Output Monitor*, which can monitor input and output transactions from the DUT separately. Listing 8 illustrates the code for the *OutputMonitor* class, which is an extension of the *uvm_component*. It accepts the name of the Cocotb proxy method as an argument, as presented in Line 2.

During the build phase, it utilizes this argument to locate the method in the proxy and subsequently invokes the method utilizing *getattr()*, as demonstrated in line 8. This functionality is infeasible in SystemVerilog-UVM. Additionally, an analysis port is created. Within the run phase, an instance of a covergroup is generated, which is sampled with the datum at each clock edge. Moreover, data is written into the analysis port, which is obtained at each clock edge, as displayed in line 14.

Listing 8: Monitor of PyUVM testbench

```
1  class OutputMonitor(uvm_component):
2    def __init__(self, name, parent, method_name):
3      super().__init__(name, parent)
4      self.method_name = method_name
5    def build_phase(self):
6      self.ap = uvm_analysis_port("ap", self)
7      self.bfm = ConfigDB().get(self, "", "BFM")
8      self.get_method = getattr(self.bfm, self.method_name)
9    async def run_phase(self):
10     dut_cg = dut_covergroup()
11     while True:
12       datum = await self.get_method()
13       dut_cg.sample(datum)
14       self.ap.write(datum)
```

E. Scoreboard

The *Scoreboard* class, detailed in Listing 9, also extends the *uvm_component* class, similar to monitors. In essence, this class receives inputs and outputs from the DUT; and analyzes and verifies the proper functioning of the DUT.

During the build phase (Lines 3-8), instances of the *uvm_tlm_analysis_fifo* are created to obtain data from monitors and store them. In the connect phase, the exports are connected, as demonstrated in lines 10 and 11. After the simulation run, the check phase is executed. Using the *try_get()* method, which returns a tuple consisting of data retrieval success (True/False) and the data, data is retrieved from both input and output FIFOs that are iterated conditionally using *can_get()*. Within the checker function as demonstrated in lines 18-22, a comparison is made to determine whether the encoder data input matches the decoder data output. Otherwise, there might be multiple bit flips (as the ECC is SECDED).

At the conclusion of the simulation, the coverage report is generated utilizing the *write_coverage_db* method from PyVSC, as shown in line 23, Listing 9.

Listing 9: Scoreboard of PyUVM Testbench

```
1  class Scoreboard(uvm_component):
2    def build_phase(self):
3      self.inp_fifo = uvm_tlm_analysis_fifo("inp_fifo", self)
4      self.out_fifo = uvm_tlm_analysis_fifo("out_fifo", self)
5      self.inp_get_port = uvm_get_port("inp_get_port", self)
6      self.out_get_port = uvm_get_port("out_get_port", self)
7      self.inp_export = self.inp_fifo.analysis_export
8      self.out_export = self.out_fifo.analysis_export
9    def connect_phase(self):
10     self.inp_get_port.connect(self.inp_fifo.get_export)
11     self.out_get_port.connect(self.out_fifo.get_export)
12   def check_phase(self):
13     while self.out_get_port.can_get():
14       _, dut_out = self.out_get_port.try_get()
15       inp_success, inp = self.inp_get_port.try_get()
16       (enc_datain, enc_chkin) = inp
17       (dec_err_det, dec_err_multpl, dec_dataout, dec_chkout, dec_alarmout) = dut_out
18       if inp_success == True:
19         if dec_dataout == enc_datain:
20           self.logger.info(f"Test: Yay!!! Passed!")
21         else:
22           self.logger.debug(f"Test: Failed! Decoded data mismatched!! Multiple bit
               flips exist!!! Expected: {enc_datain}, Actual: {dec_dataout}")
23     vsc.write_coverage_db('cov.xml')
```

F. Driver

Listing 10 illustrates the implementation of the PyUVM driver component. The *Driver* class extends the *uvm_driver* and works with sequence items. The ECC BFM method is accessed using ConfigDB. Therefore, the *get()* method is implemented in the connect phase, as shown in line 3.

In the run phase (Lines 5-9, Listing 10), the initialization function from the ECC BFM class is invoked, and then the *get_next_item()* method, which is defined inside an infinite loop, is utilized to retrieve the sequence items and transmit them to the *driver_bfm* function in the BFM class by calling *send_inp*.

Listing 10: Driver of PyUVM testbench

```
1  class Driver(uvm_driver):
2    def connect_phase(self):
3      self.bfm = ConfigDB().get(self, "", "BFM")
4    async def run_phase(self):
5      self.bfm.initialization()
6      while True:
7        input = await self.seq_item_port.get_next_item()
8        await self.bfm.send_inp(input.datain)
9        self.seq_item_port.item_done()
```

G. Sequence

The *EccSeq* class, detailed in Listing 11, extends the *uvm_sequence* class. It contains a body that generates sequence items, randomizes them, and transmits them to the Driver. Since the *await* keyword is utilized, the *start_item* and *finish_item* methods wait for access to the sequencer and transmit the items to the driver, respectively. The number of transactions can be specified in the loop count, as demonstrated in line 3.

Listing 11: A sequence in PyUVM testbench

```
1  class EccSeq(uvm_sequence):
2    async def body(self):
3      for i in range(30000):
4        in_tr = EccSeqItem("in_tr")
5        await self.start_item(in_tr)
6        in_tr.randomize()
7        await self.finish_item(in_tr)
```

H. Sequence Item

Listing 12 shows code for *EccSeqItem* which extends *uvm_sequence_item*. In the listing, lines 5-7 show that it includes all the stimuli declaration using PyVSC as specified by *@vsc.randobj* decorator at the top of class definition.

The *__eq__()*, *__str__()* methods are defined to compare and print string version of items respectively as shown in lines 9-13, Listing 12.

Listing 12: Sequence item defined in PyUVM testbench

```
1  @vsc.randobj
2  class EccSeqItem(uvm_sequence_item):
3    def __init__(self, name):
4      super().__init__(name)
5      self.correct_n = vsc.bit_t(1)
6      self.datain = vsc.rand_uint32_t()
7      self.chkin = vsc.bit_t(7)
8      ...
9    def __eq__(self, other):
10     same = self.correct_n == other.correct_n and self.datain == other.datain and ...
11     return same
12   def __str__(self):
13     return f"{self.get_name()} : correct_n: {self.correct_n}}"
```

V. RESULTS

The verification testbenches for the aforementioned design IPs are implemented using SystemVerilog-UVM and PyUVM. Specifically, PyVSC library is used along with PyUVM which enables constrained randomization and functional coverage constructs [9] for Python testbenches. The results produced using both methodologies are analyzed, compared with respect to simulation performance and features.

A. Feature Comparison

In our work, certain features are compared between SystemVerilog-UVM and PyUVM implementations as shown in Table III. Additionally, it is found that the design hierarchy for the PyUVM testbench does not include the top testbench in the simulator tools which somehow hinders its debugging capabilities. But SystemVerilog-UVM includes the top module, *uvm_test_top* along with its inner components in the simulation or debugging tools. The detailed explanation is also discussed in the work [7].

TABLE III: Comparison between SystemVerilog-UVM and PyUVM

Feature	SystemVerilog-UVM	PyUVM	Remarks
Utility Macros	`uvm_object_utils, `uvm_object_utils_begin, `uvm_object_utils_end	-	In SystemVerilog-UVM, macros are used to register classes in the UVM factory, whereas in PyUVM, All classes are already registered.
Field Macros	`uvm_field_*	-	In SystemVerilog-UVM, macros implement methods like do_compare, convert2string(). In PyUVM, __str__() and __eq__() are used for converting to string and comparing respectively.
UVM RAL Model	Provides standard base class libraries	-	In PyUVM, the implementation of RAL model is in pipeline.
Logging	Implements UVM Reporting System	Uses logging module	In PyUVM, reporting method adds two extra logging levels: FIFO_DEBUG (5) and PYUVM_DEBUG (4)
ConfigDB	uvm_config_db_#int :: set (null, "*", 'BFM',bfm)	ConfigDB().set(None, "*","BFM", bfm)	Instead of static function calls and long incantation with various types, PyUVM provides a singleton that stores data in the configuration database using the same hierarchy-based control as the SystemVerilog version. Also, PyUVM eases debugging of ConfigDB().
Importation	import uvm_pkg::*	from pyuvm import *	All the UVM classes and functions become available in our code without referencing the package.
Instantiating objects using the factory	env = alu_env::type_id:: create("env", this)	self.env = AluEnv.create ("env",self)	In PyUVM, there is no type involved as it directly copies the handles.
uvm_subscriber class	class uvm_subscriber #(type T =int) extends uvm_component	class Subscriber (uvm_analysis_export)	In SystemVerilog-UVM, uvm_subscriber creates an analysis_export with the correct parameterized type. In PyUVM, since Python does not have typing issues, a subscriber can be created by directly extending uvm_analysis_export and providing the write() function.
User-defined phases	Possible	Not possible	PyUVM only implements common phases of UVM specification.
uvm_test	Required	Not required	PyUVM does not need uvm_test, though it implements it to follow the standard with an empty extension. In run_test, class can be passed directly.
Awaiting tasks	-	await run_test()	In SystemVerilog-UVM, there is no indication for time-consuming calls, whereas in PyUVM, such calls are done using await.
TLM System	*_imp classes	-	In SystemVerilog-UVM, the *_imp classes are required to provide implementations of tasks such as put() and get(). In PyUVM, any uvm_component can instantiate uvm_put_port or uvm_get_port in its build_phase().

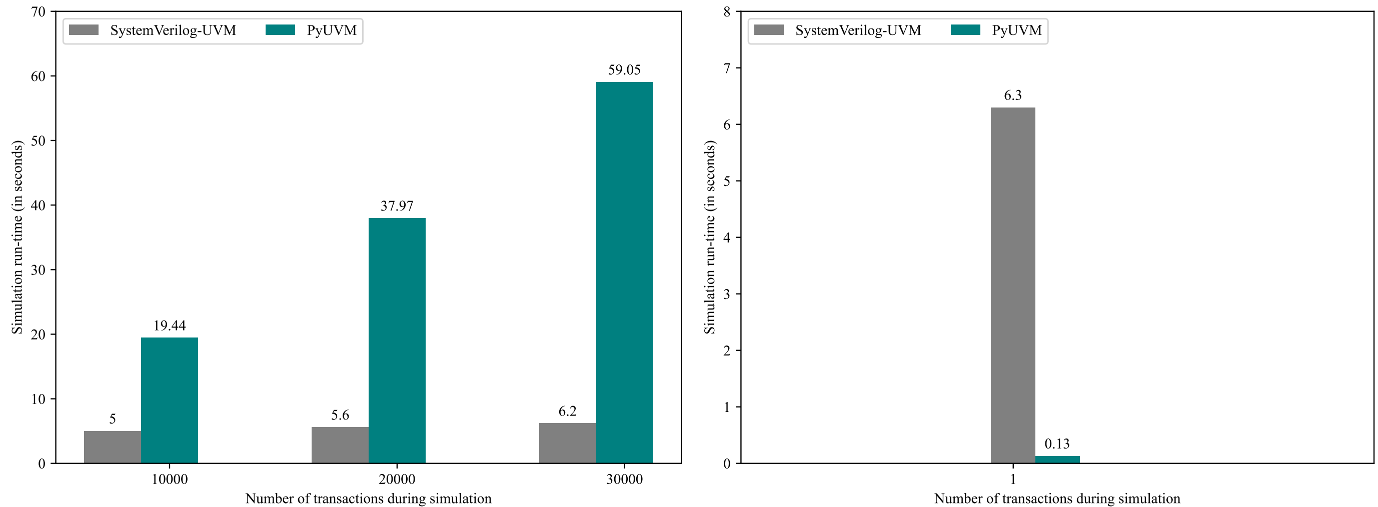
B. Performance Metrics

1) Simulation Runtime: The PyUVM and SystemVerilog-UVM testbenches are compared in terms of simulation runtime with simulator Cadence Xcelium. For both ALU and ECC, a single test specified in the testbenches is simulated with various iterations i.e., 10000, 20000, and 30000 as shown in Fig.7a and Fig. 7c. ADC testbenches include 3 tests, namely

test_feature_adc, *test_feature_reg*, and *test_stress_adc*. It is also ensured to keep all the simulation set-up parameters constant for both PyUVM and SystemVerilog-UVM testbenches to make a fair comparison.

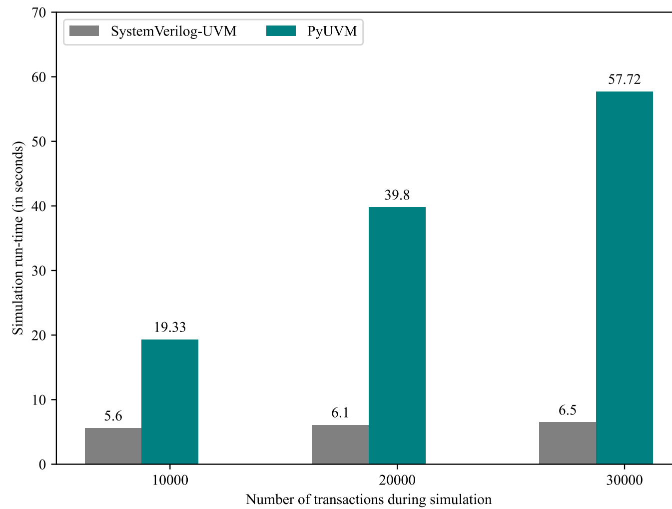
Fig.7a and Fig.7c show that the simulation run-time of PyUVM testbenches is slower than that of SystemVerilog-UVM testbenches for all iterations. This contrast in run-time performance between SystemVerilog-UVM and Python testbenches is due to their distinct approaches. With SystemVerilog, simulation directives and commands are employed to establish communication with the simulator, resulting in a close integration that enhances execution and shortens run-time. In contrast, Python testbenches interact with the simulator using VPI/VHPI, which is slower and less tightly integrated. This overhead becomes more significant as the number of transactions increases, leading to longer simulation run-time for PyUVM testbenches.

In contrast, PyUVM performs better than SystemVerilog-UVM in terms of simulation run-time for ADC as demonstrated in Fig.7c. PyUVM which uses Cocotb, automatically discovers all tests defined with the help of the *@pyuvm.test()* decorator. Therefore it may help speeding up the simulation which is not the case in SystemVerilog-UVM.



(a) ALU: A test is run separately for various transactions

(b) ADC: 3 tests are run in a single transaction



(c) ECC: A test is run separately for various transactions

Fig. 7: Simulation run-time between PyUVM and SystemVerilog-UVM

2) Coverage Analysis: To have a fair comparison between PyUVM and SystemVerilog-UVM, testbenches for previously mentioned IPs are implemented in SystemVerilog-UVM with the same coverage model. The model is defined as such that it covers all the functionalities of the design IPs as discussed in Table IV.

Our study demonstrates the feasibility of creating coverage models in PyUVM testbenches using the PyVSC library in comparison to SystemVerilog-UVM. However, there is no evidence to suggest that PyUVM outperforms SystemVerilog-UVM in terms of coverage closure. Nonetheless, the PyVSC library could be employed to restrict the stimuli and close the coverage

TABLE IV: Coverage model definition for the designs

Design IP	Cover Group(s)	Cover Point(s)	Number of Bins	Description of Cover Bin(s)
ALU	cg_1	a	5	Value ranges (-2147483648, -1768769053), (-1768769052, -866), (-865, 866), (867, 1300000000), (1300000001, 2147483647)
		b		Value ranges (-2147483648, -1654895901), (-1654895902, -989), (-988, 0), (1, 1928710300), (1928710301, 2147483647)
		op	8	Cover all 8 operations
		aXb	-	-
		bXop		
		aXop		
		aXbXop		
ADC	cg_1	en_in	2	ADC module is ON (1), OFF (0)
		rw_in	2	Register Write (1), Read (0)
		addr_in	5	Value ranges from 0 to 255
		data_in		
		ana_in		Scope: -10V to10V
	cg_2	start_out	2	Conversion is started or not
		busy_out		Transaction is ongoing
		eoc_out		Conversion is ended or not
		err_out		Covers if there is errored conversion
ECC	cg_1	data_out	8	Value ranges (0, 2975706), (2975707, 10295960), (10295961, 56784980), (56784981, 130000000), (130000001, 78939421), (789394220, 1248579698), (1248579699, 2000000000), (2000000001, 2147483647)
		chkout	3	Cover ranges (0, 23),(24, 89), (90, 127)
		err_detect	2	Error is detected
		err_multpl		Multiple error exists
		err_detectXerr_multpl	-	-

of the DUT. The results of our simulations indicate that a $100\,\%$ coverage in PyUVM simulations of the ALU design IP may have resulted from the random seed selected, as demonstrated in Table V.

TABLE V: Coverage analysis between SystemVerilog-UVM and PyUVM

Design IP	**ALU**		**ADC**		**ECC**	
SV-UVM/ PyUVM	**SV-UVM**	**PyUVM**	**SV-UVM**	**PyUVM**	**SV-UVM**	**PyUVM**
Number of Distinct Tests	1	1	3	3	1	1
Number of Transactions	30000	30000	-	-	30000	30000
Coverage (%)	78.29	100	100	100	95	95

C. Empirical Observations

While implementing the verification enivironment for the design IPs i.e., ALU, ADC, and ECC with PyUVM, there are some observations made as listed below.

1) BFM-based class approach: In SystemVerilog-UVM, an interface is used to send transactions/packets to the DUT. The declaration of stimuli inside the interface is necessary to accomplish this. On the other hand, PyUVM uses a BFM-based approach for communication between the DUT and testbench, which utilizes Cocotb. Therefore, no additional stimuli declaration is required.

2) Register Abstraction Layer Modeling: In SystemVerilog-UVM, the *UVM-RAL* models and abstracts registers and memories of a DUT. In our work, one of the design IP i.e., ADC, includes a register block with 3 registers as explained in Table II. Because of unavailability of the *UVM-RAL*, constrained randomization in the PyUVM testbench requires additional efforts to read and write to these registers, whereas in SystemVerilog-UVM, this process is simplified through the use of *UVM-RAL*. It should be noted, however, that RAL is mentioned as ”under development” as listed in Table III.

3) Ease of Testbench Development: As explained in the feature comparison, in contrast to SystemVerilog-UVM, PyUVM does not need a class constructor. Additionally, SystemVerilog does not allow introspection whereas PyUVM enables it for better coding style. For instance, the Monitor takes a proxy method name as an argument during instantiation. In the run phase of the monitor, the data can be transferred using *get_method()*, detailed in subsection IV-D. Overall, PyUVM needs less code lines compared to SystemVerilog-UVM.

4) Continuous Assignment: The testbench implementation of ECC core needs to continuously assign encoded data from the encoder to the decoder as input. Using the *assign* keyword in SystemVerilog-UVM, the assignment can be accomplished as shown in Listing 13. On the other hand, during the PyUVM testbench implementation of ECC, it is found that *data_out* is not assigned correctly with the command as mentioned in the listing 14. Consequently, instances of Encoder and Decoder are separated and encoded data is sent from the testbench instead of continuously assigning with respect to each clock edge.

Listing 13: Continuous assignment in SystemVerilog

```
assign data_input = {vif_encoder.dataout, vif_encoder.chkout};
```

Listing 14: Continuous assignment in Python

```
data_input = self.dut.enc_dataout.value.binstr + self.dut.enc_chkout.value.binstr
```

5) PyVSC usage: The coverage construct is already available in SystemVerilog. On the other hand, Python as HVL does not have a covergroup. Hence, PyVSC is used for constrained randomization and writing coverage constructs. It allows saving the coverage results in a *.xml* or *.libucis* format, which can be visualized using *PyUCIS-Viewer* [10].

VI. Conclusion

In this study, we have developed Python-based verification testbenches for three design IPs: ALU, ADC, and ECC. The testbenches have been implemented by employing the PyUVM framework and PyVSC library. The comparison between PyUVM and SystemVerilog-UVM has been carried out in terms of various features. Furthermore, the performance of PyUVM testbenches has been evaluated and compared with SystemVerilog-UVM testbenches for the aforementioned designs. Despite taking longer to run, PyUVM simulation may be more efficient as compared to SystemVerilog-UVM, provided that the clock generation is moved from the testbench to the DUT side. PyUVM simulation enabled us to conveniently collect input data along with coverage data in a preferred format. This data could be analyzed to create new methodologies based on Machine Learning techniques, which will further enhance the design verification process.

References

[1] J. Elfström, *Language Specification Length?* August 12, 2013 (Accessed: January 15, 2023). [Online]. Available: http://www.fivecomputers.com/language-specification-length.htm.

[2] H. Foster, "2022 Wilson Research Group Functional Verification Study," Siemens Digital Industries Software, Tech. Rep., Oct. 2022.

[3] M. DSU, *PY-UVM Framework for RISC-V Single Cycle Core*, https://github.com/merldsu/PY_UVM_Framework/tree/main, May 8, 2023 (Accessed: August 4, 2023).

[4] I. Quinn, "Constrained Random Stimulus Generation using Python," DVClub Europe, 2021.

[5] D. Aich, *Open-Source Python based Hardware Verification Tool*, Master Thesis, 2021. [Online]. Available: https://digital.library.txst.edu/items/a82447f4-6517-4fe8-b1b8-75d473b7e6d1.

[6] T. Poikela, *UVM-Python: UVM library for Python*, https://github.com/tpoikela/uvm-python, 2023.

[7] D. N. Gadde, S. Kumari, and A. Kumar, "Effective Design Verification – Constrained Random with Python and Cocotb," DVCon Europe, 2023.

[8] R. Salemi and T. Fitzpatrick, "Verification Learns a New Language: – An IEEE 1800.2 Implementation," 2021.

[9] M. Ballance, *PyVSC: SystemVerilog-Style Constraints, and Coverage in Python*, https://github.com/fvutils/pyvsc, 2019.

[10] M. Ballance, *PyUCIS-Viewer*, https://github.com/fvutils/pyucis, 2022.