UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Telecommunications Engineering

Evaluation and application of new Python-based frameworks for the verification of digital integrated circuits

End of Degree Project

Bachelor's Degree in Telecommunication Technologies and Services Engineering

AUTHOR: Amutio Duarte, Alejandro

Tutor: Monzó Ferrer, José María

External cotutor: GERTH, STEPHAN

ACADEMIC YEAR: 2023/2024

Resumen

El proceso de verificación de circuitos integrados evita la propagación de errores desde la etapa de diseño hasta el proceso de fabricación y el producto final, evitando así un gasto baldío de tiempo y recursos económicos. El objetivo principal es validar el comportamiento de un circuito ante cualquier entrada posible, lo que requiere la aleatorización de estímulos y, a su vez, detallados modelos de referencia. Por lo tanto, el desarrollo de bancos de pruebas se convierte en una tarea compleja. En este contexto, los ingenieros de Robert Bosch en Dresden han considerado aplicar el uso de una nueva tecnología basada en Python, con el propósito de acelerar el proceso de verificación sin afectar la calidad y la validez de las pruebas. El presente trabajo aplica un nuevo paradigma en la industria de la verificación digital basado en el lenguaje de programación Python, contrastándolo con el enfoque ampliamente adoptado de SystemVerilog. El objetivo principal del desarrollo se centra en la librería "coroutine based cosimulation testbench" (cocotb), y su implementación de la Metodología de Verificación Universal (UVM), PyUVM. La intención final es proporcionar información exhaustiva sobre su funcionamiento y uso en entornos industriales, comentando las ventajas y limitaciones respecto a metodologías alternativas. El proyecto comienza con un análisis meticuloso del panorama actual de la verificación digital, el cual abarca un examen de las metodologías existentes y una evaluación de los desafíos que cada una de ellas presenta. Posteriormente, se realiza una exploración de las librerías cocotb y PyUVM, describiendo su mecanismo operativo y desarrollando en el manejo de ambas desde una perspectiva de usuario. Para demostrar su eficacia, estas librerías se emplean en tres escenarios de verificación distintos, entre los que destaca la verificación de un diseño digital perteneciente a un circuito integrado actualmente en desarrollo. La culminación del trabajo está marcada por un resumen de los resultados obtenidos y su comparación con el enfoque convencional UVM/SystemVerilog. El segmento final evalúa críticamente la viabilidad y potencial integración generalizada de esta nueva tecnología en la industria contemporánea de la verificación digital.

Resum

El procés de verificació de circuits integrats evita la propagació d'errors des de l'etapa de disseny fins al procés de fabricació i el producte final, evitant així una despesa erma de temps i recursos econòmics. L'objectiu principal és validar el comportament d'un circuit davant qualsevol entrada possible, la qual cosa requerix l'aleatorització d'estímuls i, al seu torn, detallats models de referència. Per tant, el desenvolupament de bancs de proves es convertix en una tasca complexa. En este context, els enginyers de Robert Bosch en Dresden han considerat aplicar l'ús d'una nova tecnologia basada en Python, amb el propòsit d'accelerar el procés de verificació sense afectar la qualitat i la validesa de les proves. El present treball aplica un nou paradigma en la indústria de la verificació digital basat en el llenguatge de programació Python, contrastant-lo amb l'enfocament àmpliament adoptat de SystemVerilog. L'objectiu principal del desenvolupament se centra en la llibreria "coroutine based cosimulation testbench" (cocotb), i la seua implementació de la Metodologia de Verificació Universal (UVM), PyUVM. La intenció final és proporcionar informació exhaustiva sobre el seu funcionament i ús en entorns industrials, comentant els avantatges i limitacions respecte a metodologies alternatives. El projecte comença amb una anàlisi meticulosa del panorama actual de la verificació digital, el qual abasta un examen de les metodologies existents i una avaluació dels desafiaments que cadascuna d'elles presenta. Posteriorment, es realitza una exploració de les llibreries cocotb i PyUVM, descrivint el seu

mecanisme operatiu i desenvolupant en el maneig de totes dues des d'una perspectiva d'usuari. Per a demostrar la seua eficàcia, estes llibreries s'empren en tres escenaris de verificació diferents, entre els quals destaca la verificació d'un disseny digital pertanyent a un circuit integrat actualment en desenvolupament. La culminació del treball està marcada per un resum dels resultats obtinguts i la seua comparació amb l'enfocament convencional UVM/SystemVerilog. El segment final avalua críticament la viabilitat i potencial integració generalitzada d'esta nova tecnologia en la indústria contemporània de la verificació digital.

Abstract

The verification process for integrated circuits prevents the propagation of errors from the design stage to the manufacturing process and the final product, thus avoiding a futile spending of time and economic resources. Ideally, the main goal is to validate the behaviour of a circuit under any inputs, which requires elaborated stimuli randomization and intricate reference models. Hence, rendering the development of testbenches a complex task. In this context, engineers from Robert Bosch at Dresden have considered the use of a new Python-based technology, with the purpose of speeding up the verification process without affecting the quality and validity of the tests. This work applies a new Python paradigm to digital verification, contrasting it with the widely adopted SystemVerilog approach. The primary focus of this project centres on the "coroutine based cosimulation testbench" (cocotb) library and the Universal Verification Methodology (UVM) abstraction of it (PyUVM). The overarching goal is to provide comprehensive insights into their practical application in industrial settings, commenting on the advantages and limitations vis-à-vis alternative methodologies. The project starts with a meticulous analysis of the current landscape of digital verification, encompassing an examination of extant methodologies and an assessment of the challenges inherent in their implementation. Subsequently, an exhaustive exploration of the cocotb and PyUVM libraries ensues, describing their underlying operational mechanism and untangling the expected usage patterns from the end-user perspective. To substantiate their efficacy, these libraries are deployed in three distinct application scenarios, notably including the verification of a digital design part of an integrated circuit currently in active development. The culmination of the present work is marked by an extensive summary, wherein the obtained results are presented and carefully benchmarked against the conventional UVM/SystemVerilog approaches. The concluding segment critically assesses the viability and potential wide integration of this Python framework within the contemporary digital verification industry.

I would like to express my deepest gratitude to Stephan Gerth and José María Monzó Ferrer, for this thesis would not have been possible without their teachings and their unwavering support. I am also thankful for the opportunity that the Mobility Electronics team at Robert Bosch has given me and for Eik Bergmann and his support in my future endeavours. Special thanks to Colin Marquardt for providing invaluable feedback and advice when I needed it.

Lastly, I would like to acknowledge my family and my closest friends, for being by my side even when I was not the person that I wanted to be. Thanks.

Table of contents

Chapter 1. Introduction ... 7
1.1 General overview ... 7
1.2 Objectives ... 8
1.3 Methodology ... 8
1.3.1 Setup ... 8
1.3.2 Metrics ... 9
1.3.3 Development timeline ... 9
Chapter 2. State of the art ... 11
2.1 Design flow ... 11
2.2 Verification plan ... 12
2.3 Static verification ... 12
2.4 Functional verification ... 15
2.4.1 Universal Verification Methodology ... 17
Chapter 3. Study of cocotb ... 19
3.1 Overview ... 19
3.2 Internal functioning ... 19
3.3 Coroutines ... 21
3.4 Basic concepts ... 21
3.5 PyUVM ... 23
Chapter 4. Synchronous First-In First-Out ... 28
4.1 Specifications and verification plan ... 28
4.2 Reference model ... 29
4.3 Testbench structure ... 30
4.4 Test results ... 32
4.4.1 Testbench comparison ... 35
4.4.2 Discussion ... 36
Chapter 5. Bus arbiter ... 37
5.1 Specifications and verification plan ... 37
5.2 Reference model ... 40
5.3 Testbench structure ... 41
5.4 Test results ... 44
5.4.1 Testbench comparison ... 51
5.4.2 Discussion ... 52

Chapter 6. Chirp Start 53
6.1 Specifications and verification plan 53
6.2 Reference model 57
6.2.1 Clocking issues 57
6.2.2 Model behaviour 57
6.3 Testbench structure 57
6.4 Test results 59
6.4.1 Discussion 62
Chapter 7. Summary 63
7.1 Comparison 63
7.1.1 Limitations and target cases 65
7.1.2 Performance enhancing 65
7.2 Goal review 65
7.3 Concluding thoughts 66
7.4 Future work 66
7.5 SDG alignment 67

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
BFM	Bus functional model
Bosch	Robert Bosch GmbH
CERN	European council for nuclear research
CRV	Constrained random verification
DUT	Device under test
EDA	Electronic design automation
FLI	Foreign language interface
FOSSi	Free and open-source silicon foundation
FPGA	Field-programmable gate array
FSM	Finite state machine
GPI	Generic procedural interface
HDL	Hardware description language
IC	Integrated circuit
ME	Mobility Electronics
ORConf	Open-source silicon conference
RTL	Register Transfer Level
SDG	Sustainable Development Goals
SFIFO	Synchronous first-in first-out
SVA	SystemVerilog assertion
UN	United Nations
UVM	Universal Verification Methodology
VHDL	Very high-speed integrated circuit hardware description language
VHPI	VHDL procedural interface
VPI	Verilog procedural interface
cocotb	“coroutine based cosimulation testbench”

List of Figures

Figure 1. Simplified ASIC design flow from market survey until success of RTL verification. 11
Figure 2. Truth table of an AND logic gate. 13
Figure 3. Possible states of a design behaviour and those allowed by specifications. 14
Figure 4. Basic multi-agent UVM testbench. 18
Figure 5. Block diagram example of C code execution in a simulator using VPI. 20
Figure 6. Block diagram on the internal functioning of cocotb. 20
Figure 7. Block diagram of the SFIFO design. 28
Figure 8. UVM structure of the SFIFO testbench. 31
Figure 9. Simulation results of the first SFIFO version. 32
Figure 10. Waveform of TstWriteFull simulation for the first SFIFO version. 33
Figure 11. Waveform of TstReadEmpty for the first SFIFO version. 33
Figure 12. Simulation results of the second SFIFO version. 34
Figure 13. Waveform of TstRandom for the second SFIFO version. 34
Figure 14. Simulation results of the third SFIFO version. 34
Figure 15. Coverage of TstRandom for the third SFIFO version. 35
Figure 16. Comparison of the mean runtime of different SFIFO testbench approaches for varying iterations of TstRandom. 36
Figure 17. Bus arbiter connection structure and address map. 38
Figure 18. State diagram of the AMBA 3 APB protocol for a master-slave access. 38
Figure 19. Flowchart for master-slave connection assignment. 40
Figure 20. UVM structure of the bus arbiter testbench. 42
Figure 21. RandomGen sequence flowchart model. 43
Figure 22. Transaction sequence weighted random stimuli. 44
Figure 23. Arbiter error regarding Master0 and Slave2. PSLVERR mismatch. 45
Figure 24. Arbiter error regarding Master0 and Slave2. No PSLVERR mismatch. 45
Figure 25. Arbiter error regarding Master2 and Slave3. Unmasked address value. 46
Figure 26. Arbiter error regarding Master2 and Slave2. PSEL mismatch. 46
Figure 27. Arbiter error regarding Master2 and Slave2. No PSEL mismatch. 47
Figure 28. Arbiter error regarding Master1 and Slave2. PSLVERR mismatch. 47
Figure 29. Arbiter error regarding Master1 and Slave0. PRDATA mismatch (MSB = 1). 48
Figure 30. Arbiter error regarding Master1 and Slave0. PRDATA mismatch (MSB = 0). 48
Figure 31. Arbiter error regarding Master2 and Slave1. PADDR mismatch. 49
Figure 32. Arbiter error regarding Master2 and Slave1. An additional PADDR mismatch. 49
Figure 33. Arbiter error regarding Master0 and Slave1. PSLVERR mismatch. 50
Figure 34. Coverage of TstRandom for Master1. 51

Figure 35. Comparison of the mean runtime of different bus arbiter testbench approaches for varying iterations of TstRandom. 52

Figure 36. Exemplary waveform of a signal not being captured in a clock domain crossing scenario. 53

Figure 37. Exemplary waveform of START being correctly translated from the *clk* domain to the *pclk* one. 54

Figure 38. Simplified block diagram of the Chirp Start system. 54

Figure 39. Exemplary waveform on how the SAMP signal must be received. 54

Figure 40. Exemplary waveform on how internal signals RS_SAMP and RS_PCLK are generated. 55

Figure 41. Exemplary waveform on how the calibration mode operates. 55

Figure 42. Exemplary waveform on how to trigger the calibration mode. 56

Figure 43. Exemplary waveform on how the relay mode operates. 56

Figure 44. UVM structure of the Chirp Start testbench. 58

Figure 45. Examples of implementation of the SAMP signal. 58

Figure 46. State diagram for the random test of the Chirp Start testbench. 59

Figure 47. Calibration measurement. Assertion error on ACTIVE. 60

Figure 48. Calibration measurement. Expanded simulation. 60

Figure 49. Calibration measurement. Correct behaviour of ACTIVE. 60

Figure 50. Calibration measurement. Accumulator values. 61

Figure 51. Chirp Start error regarding missing RAMP signal after a reset. 61

Figure 52. Chirp Start error regarding missing RAMP signal after a calibration. 62

Figure 53. Mean runtime of TstRandom across DUTs. Normalized to the respective slowest implementation. 63

List of Tables

Table 1. Line count comparison for SFIFO testbenches. 35

Table 2. Line count comparison for bus arbiter testbenches. 51

Chapter 1. Introduction

1.1 General overview

Developing a new *integrated circuit* (IC) is a resource-intensive task that can expand indefinitely through time as well as through economical capabilities. These ICs present more complexities with each iteration, allowed by the advance in semiconductor technology, and a manufacturing failure of a sole product can cause havoc for any company. Errors, malfunctions, or mismatches between the expected behaviour and the one observed, should be corrected before it propagates further down the design workflow, avoiding the fabrication of a faulty design at all costs.

A study from 2020 (1) notes that the growth in demand for verification engineers between 2007 and 2020 was more than double that for design engineers. Although the mean ratio between both positions seems to be one-to-one across the industry, specific markets (for instance, processor units) may experience 5-to-1 ratios. Verification is not merely a quality control measure; it is the linchpin that fortifies the foundation upon which the semiconductor industry thrives.

Digital designs are built using a *Hardware Description Language* (HDL) such as Verilog (2) and VHDL (3) to transform lines of code into synthesizable logic. However, this is not a requirement for verification, and that allows other languages to thrive. Eventually, a Verilog superset, SystemVerilog (4), was created to unify design and verification in a single language, which in turn has synthesizable structures and others that are not.

SystemVerilog has been the most popular option for digital verification (5) in the last decade. Nonetheless, as ICs grow in complexity, the tests and the models also do. This intricacy presents a problem for a language such as SystemVerilog. Its syntax complexity does not provide the ideal environment for maintainability, and its non-usage outside the semiconductor industry introduces a significant barrier for new verification engineers.

Python (6) is one of the most used programming languages across disciplines with more than 512,937 projects available (7), and *cocotb* (coroutine based cosimulation testbench) (8) takes advantage of it. First released in 2013 (9), the cocotb framework allows for integration into an elevated number of the available HDL simulators, enabling the creation of testbenches using the Python language. Its popularity has awarded it to be used in the industry, presented at verification conferences (10) , and the creation of libraries that build upon it, the most notable being *PyUVM* (11).

This project is a result of the imperative for enhancing verification methodologies, specifically for digital designs. Semiconductor companies, such as *Robert Bosch GmbH* (Bosch) being the director of this thesis, have a special interest in cocotb. This document will explore this library and related frameworks, thus providing helpful insight into the potential benefits that this emerging technology brings to the forefront of IC digital verification. Its comparison to already known and established verification methodologies, and its application to an in-development industrial design, will help to determine its significance in future projects. Additionally, this dissertation may help to understand how to apply cocotb in further verification environments.

1.2 Objectives

The main objective of this project is to apply the cocotb framework as a behavioural digital verification tool and extract how this technology can be further integrated into (or replace) existing verification environments based.

Significant milestones will define the success of achieving the main goal:

- Examine the current state of the IC verification industry, the challenges of SystemVerilog-based testbenches, and how cocotb can address them.
- Provide a straightforward understanding of how cocotb works, as well as how to use it for testbench design.
- Design and run cocotb-based testbenches to verify diverse digital designs under controlled scenarios: a ramp-up project and a significantly more complex design.
- Evaluate how cocotb-based testbenches compare to SystemVerilog ones using the previously verified designs.
- Apply the acquired knowledge of cocotb to verify a digital design in active development by Bosch.
- Explore how cocotb can be used for future projects inside the company and, more generally, in the verification industry.

1.3 Methodology

This project was conducted at the facilities of Robert Bosch GmbH in Dresden, Germany, under the guidance of the *Mobility Electronics* (ME) department. Certain set-up decisions have been selected according to company standards or by common usage across the department.

1.3.1 Setup

The computer environment in which simulations are run has been configured according to the cocotb requirements specified in its documentation.

- **Operative system**: The selection of a UNIX-based OS (*Fedora 7.3*) is a Bosch decision. The work environment is already set up, which includes the simulator software and licenses to use it.
- **cocotb**: The latest release available while developing this thesis (v.1.8.0) has been chosen. It provides an updated insight into the current capabilities and limitations of the framework.
- **Python**: The used version must be no older than 3.6. To maximize the benefits of the language, the most recent release with ongoing security maintenance until October 2026 (v3.10) has been chosen. This ensures that written code will remain compatible with future updates.
- **Code editor**: Among the diverse options for code editors, *Visual Studio Code* was the preferred choice due to its intuitive usage and abundance of external plugins to enhance the user experience. The used version corresponds to build number 1.82.2.
- **GNU Make**: To effectively execute the Python code with system calls to the simulator, an executable is essential. The framework requires *GNU Make* as the chosen tool, and it requires a version no earlier than the third release (v3) to ensure compatible functionality. The given environment provides the 4.3 version.
- **Simulator**: Given that the framework only provides the means to code the testbench, a compatible simulator must be used to execute it. Due to the abundance of licenses available in the workspace, the ME team opted to use *Cadence Xcelium*. This decision was influenced by their previous testbenches being successfully executed using the same simulator. The proximity in experimental conditions will enable a meaningful comparison of the results obtained for both approaches.

Three designs have been chosen for this project, all of them provided by Bosch. Each one represents a *Design Under Test* (DUT) to be verified:

- ***Synchronous First-In First-Out* (SFIFO)**: Ramp-up project for verification engineers. Since it is the first DUT for new employees, it provides helpful data regarding the learning curve of the framework.
- ***Bus arbiter***: Verification challenge from 2018. It presents a more complex design that requires a more demanding testbench, therefore highlighting the benefits and limitations of cocotb against other approaches.
- ***Chirp Start***: Digital design block found in the current project being developed by the ME team. It will set the stage for future real-life usage cases of the framework in the company.

1.3.2 Metrics

To perform a valid comparison against already existent verification environments, the conditions of testing must be similar. Although different companies may have diverse approaches to digital verification, the one used the most across the field (5) is the *Universal Verification Methodology* (12) (UVM). Therefore, it has been chosen as the standard for this comparison.

The UVM approach to verification is discussed and explained in section 2.4.1. However, it can be described as a workflow/framework to implement reusable testbenches.

For SystemVerilog, there exists a library to implement this framework, while for cocotb there are multiple libraries available. During this project, the PyUVM library will be used as the default implementation of UVM in cocotb. This selection against other alternatives is discussed further ahead.

Henceforth, the application of UVM in SystemVerilog will be referred to as *SV/UVM*. To evaluate how it differs from PyUVM, the comparison will focus on the following categories:

- **Learning curve**: How easy it is to start a project, increase knowledge, and master its use. Availability of documentation, tutorials, and community support will be considered.
- **Performance**: The simulation speed of each of the testbenches. This metric can vary depending on the efficiency of the written testbench rather than the simulator used. Additionally, the number of code lines between testbenches will be compared, although the same statements could have been expressed differently.
- **Integration**: Ease of integrating each of the options with existent and future verification tools.
- **Maintainability**: How the code is structured and how the programming language used can influence the maintenance of the testbench.
- **Industrial adoption**: Number of users, successful deployments, and industry recognition.

1.3.3 Development timeline

To accomplish the target objectives, the project has been structured into specific tasks. Each task has been assigned a time frame in which it should be completed. Note that the development of this thesis extends from September 2023 until March 2024.

1. **cocotb basics**: This part of the project involves setting up the work environment as well as designing and running basic testbenches using the cocotb and PyUVM frameworks. A month is dedicated to this purpose.
2. **SFIFO and bus arbiter verification**: This task is related to the development of PyUVM and SV/UVM testbenches for both the SFIFO and the bus arbiter designs. Being the first application of the framework and involving multiple DUTs, this development phase takes place over two months.
3. **Chirp Start verification**: The verification of a real design requires special dedication. The functioning of both PyUVM and cocotb is already known, but the testbench for this

scenario requires higher attention to detail. Thus, a month is the time given to complete this task.

4. **Summary and writing**: The analysis of the results, as well as narrating the development process, is a complicated task that involves constant review between the people involved in this thesis. Therefore, two months is expected to coherently assemble the recorded notes into this document.

Chapter 2. State of the art

Electronic designs in the digital domain contemplate electronic circuits whose input and output signals are discrete values, governed by Boolean logic. As a simpler definition; circuits whose minimal building blocks are logic gates.

This chapter aims to provide the reader with background knowledge on the *Register Transfer Level* (RTL) design of digital *Application-Specific Integrated Circuit* (ASIC), with special attention to the techniques used in the process of verification.

2.1 Design flow

When building a new system, each company may have its own custom approach to the design flow to follow. Nonetheless, these processes are mostly based on the same basic steps [13] [14] [15] [16]. For simplicity, the market survey phase and stages regarding physical design will be omitted.

The process starts with a design idea, and at the same time, the specification phase takes place. During this step, the design architecture is defined, and its behaviour and characteristics are described as detailed as possible. Concurrently, an attempt at balancing system requirements (based on client targets) and capabilities happens. The main goal of this phase is to establish how the overall system and its architectural blocks behave.

Although specifications are the basic description of a design, new scenarios and applications can be discovered once the design process has taken place. As a result, specifications can be improved in further stages and are constantly open to modifications.

Once the specifications have a solid foundation, the RTL design phase can commence. To describe digital logic as well as memory components, designers use a dedicated language. An HDL may look like a common programming language, but it can characterize logic components as readable text. Popular options are Verilog, SystemVerilog, and VHDL. As soon as all RTL designs are finished, they are merged into a final top design.

Concurrently to the design stage, the verification process takes place. Using different techniques, the design behaviour is compared against the specification. This process has a close connection to the RTL design step. As verification engineers discover faulty functioning in the design, the results are reported to the designer to be fixed. Once the system behaves as expected, the next stages can take place.

Further steps on the design flow include synthesizing the HDL code into logic structures, describing the connection between these structures (*netlist*), and the fabrication of the physical layout.

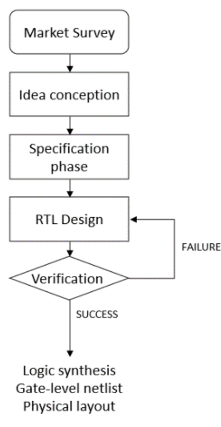

Figure 1. Simplified ASIC design flow from market survey until success of RTL verification.

The existence of the verification process is meant to avoid the downstream propagation of errors and faulty behaviours of the design. This avoids the manufacturing of partially defective circuits, which can result in a significant economic impact on a company. In addition, backtracking the design flow from the manufacturing phase until the RTL design has a notable time impact, thus resulting in release delays and an overall disadvantage over competitors. In an industry where 49.3% of the market revenue corresponds to just ten companies (17), pre-silicon verification becomes an asset.

These verification engineers, although closely related to the design team, are a separate entity whose object is to compare a design against its specifications. The designers should not execute this task themselves as they inevitably introduce a level of bias to the process.

The designers have explicit knowledge on how the HDL code is going to behave, then they could inadvertently avoid testing specific cases, or only check specific scenarios where the design works correctly. An external observer provides valuable information, as it will determine if the specifications are unambiguous and if the design behaves accordingly.

The importance of the verification process is remarkably significant that in some market sectors, for instance, microprocessors, it is possible to find five verification engineers per designer (1).

2.2 Verification plan

For the verification process, there are multiple tools available. The main target, however, is to check that the design behaves accordingly, that it can work for every expected circumstance, and that it can manage unexpected scenarios correctly. Even though there are multiple approaches to evaluate the design's capabilities, a common step is defined: a verification plan.

This phase has a significant importance, as it lays out the failure and success conditions. If the verification plan is completed successfully, then the design is stated to work correctly and can continue the development flow. If not, the errors are reported to be fixed.

The plan introduces details such as to which verification approach or methodology to follow, which cases are forbidden by the specifications, which cases may present special interest, and how much the design should be stressed. This last feature establishes how many input combinations should be tested, thus indirectly designating for how long the test should run.

In the end, this is a delicate stage where every detail must be drawn, but it is also the most crucial. The verification plan is responsible for determining the conditions and scenarios that have significant importance to assess the validity of a design.

While there are multiple methodologies to approach the verification of a design, the following sections will focus on two of the most significant ones (18); *static* and *functional* verification.

2.3 Static verification

Digital designs are based upon logic elements and thus, they can be described using Boolean arithmetic. This factor allows digital designs to be characterized as mathematical models, which introduces the possibility of assessing their validity (also referred to as *correctness*) with a mathematical approach.

Take for instance the truth table of Figure 2 defining the functioning of an AND gate. It specifically defines the expected output for a given set of inputs. If a verification engineer wanted to prove that every time both inputs are 1 the output is also 1, it could use equation (1) to mathematically assert that statement.

$$\mathbf{A \cdot B = C} \quad \mathbf{(1)}$$

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2. Truth table of an AND logic gate.

This mathematical approach to verification is given the name of static verification, although it can also be referred to as *formal* verification. This methodology requires specific tools that may differ from what the reader may expect.

In essence, a formal verification tool analyses the provided HDL code and extracts its logical behaviour, being able to describe it as a mathematical model. In turn, this allows the tool to handle every possible state of the design.

The behaviour of a design can be checked as a set of logical properties that must be asserted. An assertion is a statement about a logical property that should hold under a set of conditions. If the statement is found to be false under those circumstances, the assertion is said to have failed.

In this context, there exist formal specification languages that let the user define assertions for a target design. These languages can be interpreted as mathematical statements that formal verification tools can use to observe if they hold against the design's mathematical model.

The *SystemVerilog Assertion* [19] (SVA) language is commonly used for this task, building upon the popularity of the SystemVerilog language itself. The language allows a variety of structures to describe the specifications of a design, although this section will only cover the basics.

An assertion is defined by the *assert* keyword and can be either immediate or concurrent. An immediate assertion is checked in reference to its location on the code; it can be considered a condensed if-else statement. A concurrent assertion, on the other hand, establishes a property to be checked constantly at any point, or state.

Both types of assertions can be used in formal and functional verification, nevertheless, both types are explained in this section. To understand concurrent assertions, it is helpful to first provide an overview of the immediate approach:

```
assert (C == 1);
CheckForC: assert (C == 1) $display("C is 1") else $display("C is not 1");
```

Note that both assertions are immediate but additional options are added to the second one. Both statements check for the value of C being 1, but the second one is assigned a name (*CheckForC*) and displays a message if the assertion succeeds or fails. In formal verification tools, the failure or success of the assertion will be reported whether there is a *$display* message or not.

Concurrent assertions may follow a similar structure. They are described by adding the *property* keyword to the assertion statement. For example:

```
CheckWrRd: assert property (!(WR && RD));
```

The previous assertion declares that at any given moment, the signals WR and RD cannot be both 1. If at some point, or state, both signals are at their high-level value, the assertion will trigger a failure.

More types of behaviours can be described using the SVA language, but their explanation escapes the scope of this section. Nonetheless, to provide further insights into the capabilities of the language, another example is shown:

```
BAfterA: assert property @(posedge clk) disable iff (!reset_n)
         $rose(A) |-> ##[1:3] $rose(B);
```

The previous assertion declares initially that the check should take place at the rising edge of the *clk* signal and only if the *reset_n* signal is not 0. Afterwards, if the signal A changed from 0 to 1 (*$rose*), at some point between the next three *clk* cycles the signal B should also change from 0 to 1.

At this point, it may be clear that the formal simulator has a mathematical model for the design (from the HDL code) and another one for the expected behaviour (SVA code), but not all scenarios may be possible to check. Specifications of a design may forbid specific input combinations, and the SVA language provides a solution, called *assumptions*.

An assumption lets the simulator know which input combination or scenarios are not allowed by the specifications, for example:

```
reset_1clk: assume property @(posedge clk) !reset_n |=> reset_n;
```

The previous assumption establishes that if the *reset_n* signal is set to 0, it must return to 1 in the next *clk* cycle. The simulator will correctly understand this assumption and constrain its input stimuli accordingly.

This approach to verification may seem ideal. The engineer can almost input the specifications as plain text and the simulator manages the rest. Nonetheless, there is a significant drawback [(11)]; runtime.

The design is expressed as a mathematical model, but it is still a combination of multiple equations with multiple outcomes that influence future equations. The tool evaluates every combination and every state transition; this is how the result of an equation affects the ones that depend on it.

Each design presents a specific number of logic combinations (*states*), and multiple paths to get to each of them. States can either be illegal or valid. Illegal states represent a faulty behaviour of the design based upon the specifications. On the other hand, valid states are logic combinations allowed by the specifications. Figure 3 provides further information on the possible states of a design.

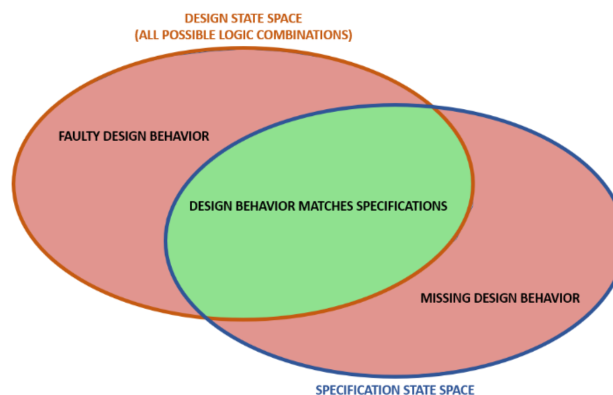

Figure 3. Possible states of a design behaviour and those allowed by specifications.

A state contained in the design space but not covered by the specifications is considered a faulty behaviour, an error. On the other side, a state contained in the specification state but that does not have a design implementation represents a missing functionality.

The goal of the formal tools is to match both the specification and the design spaces using assertions. These tools have different approaches to checking all these states and transitions (bounded model check, induction, etc.), based on mathematical techniques. This process requires a noticeable runtime the more complex a design is; the more logical states are introduced.

For basic/simple designs, formal verification may be the best approach as it provides a straightforward case to completely verify its behaviour. For more complex designs, the number of available states and state transitions may make the simulation run for longer, thus making it unsuitable for an agile workflow. Nonetheless, formal verification is a valid methodology being used in the industry (18) and, like other alternatives, it has its target cases and its disadvantages.

2.4 Functional verification

A functional approach to verification refers to handling the inputs and outputs of the DUT directly through a simulator. It differs from formal verification approaches as the target is not to check the overall correctness of a design but rather to observe how it behaves under certain stimuli. The elevated runtime of formal verification makes this approach an attractive option for complex sequential designs, or to speed up the verification process when checking specific scenarios.

Referring to Figure 3, functional verification has the goal of matching both the specification and the design spaces using custom stimuli.

There are specific tools that can synthesize and extract the behaviour of an HDL design to simulate it; HDL simulators. The design functionality is understood, and the simulator can imitate its behaviour using computer operations. In this case, the design is not parsed into a mathematical model but rather into a set of executable instructions.

Simulators per se do not present the autonomous capability to analyse the correctness of a design, the engineer must generate the stimuli. To manipulate the inputs and extract information from the outputs, a testbench is used. A testbench (also shortened to *tb*) is a structure written in an HDL that handles the stimulus generation as well as the output checking. The DUT module is instantiated and its signals are controlled directly. Engineers determine the stimuli to use based on the cases or scenarios to evaluate the test cases.

Although there exist multiple options of HDL to perform functional verification, this section will explore the SystemVerilog language, based on its popularity for this approach (18).

The following code presents a simple testbench module. Internal signals are created to connect them to the ones in the DUT, which is instantiated afterwards. Initially the value of *d* is set to 1, and after the next rising edge of the *clk* signal, the value of the *q* signal is checked to be 1.

```
module tb_example;
    reg d, clk, q, q_n;

    flip_flop_d dut (.d(d), .clk(clk),
                     .q(q), .q_n(q_n));
    initial begin
        d <= 1'b1;
        @(posedge clk);
        assert (q == 1'b1);
    end
endmodule
```

Functional verification allows engineers to customize the stimuli generation step by step. If static verification checks every possible state, functional verification could directly observe the behaviour of a design for tailored scenarios.

However, testing how a design works under a constant input sequence endlessly does not provide enough information. Ideally, the input must represent a real usage situation; different scenarios in a varying sequence.

Randomized stimuli approximate the real conditions under which the design will be held. For instance, in the previous example, the value of the *d* signal was explicitly set to 1. However, that may not be always the case, as the same signal could have also been 0. It is possible to add a couple more lines of code to also test this last case, although it is not a scalable method for signals with a higher number of bits.

The idea of randomized stimuli is a powerful tool that may require control over the generated values. The specifications of a design may describe not allowed input combinations, sequences, or usage conditions. In these cases, signals may have explicit rules to obey or a range of valid values. *Constrained Random Verification* (CRV) allows for a wide range of possible scenarios as the inputs of the DUT are constantly randomized, while also targeting the functionality of a design. This approach allows for input signals to take different values through the simulation, while at the same time limiting the valid values by specific conditions.

In SystemVerilog, random variables contained inside a class are defined with the *rand* keyword. Their value is randomized when calling the *randomize* function of the class. On the other hand, constraints are defined with the *constraint* keyword and can implement different types of restrictions, from simple ones to more intricate scenarios. An example follows:

```
class crv_class;
    rand byte in_data;
    rand bit reset_n;

    constraint allowed_range {in_data inside [2:5];}
    constraint reset_val {reset_n dist {0:=80, 1:=20};}
endclass

crv_class tst_example = new();
tst_example.allowed_range.constraint_mode(0);
tst_example.randomize();
```

The previous code shows a class (*crv_class*) containing two randomizable variables: *in_data* and *reset_n*. Then, two constraints are defined. The first one limits the *in_data* variable to values between 2 and 5 (both included). The second one establishes a weighted distribution of values for *reset_n*: the value will be 0 with a probability of 80/100, and it will be 1 with a probability of 20/100. Afterwards, an instance of the class is generated (*tst_example*) and before its variables are randomized, the constraint regarding *in_data* is deactivated.

This randomized approach to verification allows for a colourful palette of combinations and sequences to be observed. Nevertheless, the simulation could run indefinitely if not stopped manually or by the failure of an assertion. To determine if the DUT has been stressed enough, the goals established in the verification plan come into use.

To assess the validity of a design, the general rule is that the more cases are evaluated the better. Static verification represents the extreme of this approach, as all possible scenarios are covered. In functional verification, however, because the stimuli are generated by the engineer, specific cases may be never tested. The solution to this problem is to measure which scenarios have taken place: the *coverage* of the testbench.

The SystemVerilog language natively implements structures to handle coverage. A *coverpoint* is an expression or statement that will be evaluated at a sampling point. A *covergroup* contains

multiple coverpoints and specifies a common sampling point for the evaluation of each of them. The percentage of coverage from a coverpoint is defined by the number of sampled cases and the number of possible ones. A simple example follows:

```
 covergroup my_coverage @ (posedge clk);
     cover_data  : coverpoint in_data;
     cover_rst   : coverpoint reset_n;
endgroup
```

The previous code establishes that for every rising edge of the *clk* signal, the values of *in_data* and *reset_n* will be registered. Each coverpoint will have a complete coverage (100%) when all their possible values are sampled; when each signal has taken every possible value.

While effective, the coverage tool even presents more capabilities. For instance; if the signal *in_data* is constrained to values between 2 and 5, its coverage will never be complete. Additionally, it may not make sense to sample the value of *in_data* if the reset is active. For these scenarios, there is a possible implementation:

```
 covergroup my_coverage @ (posedge clk);
     cover_data : coverpoint in_data iff (reset_n) {
         bins my_range = {[2:5]};
     }
endgroup
```

In the previous code, the value of *in_data* will only be sampled in the range between 2 and 5 and whenever the reset is not active (high-level). More coverage constructs are available to use, such as observing the combinations between two signals (*cross* coverage), but they present a more advanced topic not related to this thesis.

2.4.1 Universal Verification Methodology

A common approach to testbenches does not present a real option for scalability and reusability. Changes and evaluation of the DUT signals are made through statements that are written directly on the testbench code. Each time a new test is written, the engineer must write the sequence to follow and, at the same time, handle how the variables are connected to the DUT. Ideally, the last step would be ignored; the engineer would only have to write the sequences to run and there would be a structure in place to send the sequence to the DUT.

The *Universal Verification Methodology* (UVM) is an approach to apply the best practices for verification, focused on the reusability of its components for changing testbenches. It was designed by *Accellera* based upon the *Open Verification Methodology* and it is now an *IEEE* standard [12] supported by multiple simulators from varied *Electronic Design Automation* (EDA) vendors. It is also the predominately used methodology for verification in industrial settings [18]. A library that implements the UVM methodology can be found in the SystemVerilog language.

In summary, this methodology is a powerful tool that can improve reusability and efficiency for testbench design. The capabilities of UVM are notably useful as they target exclusively the improvement of the verification process. In turn, mastering UVM is a time-consuming task, and learning the basics of it also presents challenges. It is a methodology with a steep learning curve that, when understood, drastically boosts testbench design and reusability.

This section will focus on a basic description of a simple UVM testbench. This technology, while widely used in the industry, presents characteristics that allow for advanced testbenches, and thus, to cover all its capabilities, entire books have been dedicated to that purpose. This section aims to provide the reader with a basic notion of how UVM testbenches operate [20].

Further explanations will be made based on the block diagram shown in Figure 4.

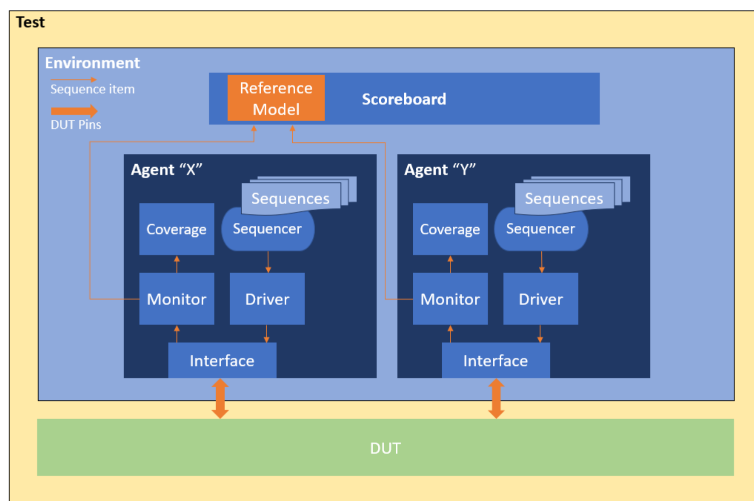

Figure 4. Basic multi-agent UVM testbench.

The minimal unit on a UVM testbench is a *sequence item*. This is the lowest object needed to send information from an engineer's perspective to the DUT. These items contain information which data shall be driven to the DUT.

A set of ordered sequence items is called a *sequence*. A sequence can be seen as a function; it contains multiple directives in the form of sequence items to interact with the DUT. To execute these directives, a *sequencer* manages when the next item of a sequence must be called. A sequencer can only handle one sequence at a time.

To interact with the DUT, UVM testbenches use a *driver* and a *monitor*. The first component receives the sequence items from the sequencer and decodes the data into the DUT input pins. On the other hand, the monitor encodes the DUT output pins into a sequence item. Each component has a specific event that triggers its functioning. For example, the monitor could execute its function on every falling edge of the clock signal, and the driver could send the next item to the DUT on every rising edge. A set of DUT pins is defined by the *interface* SystemVerilog structure

All these components are wrapped by a class called *agent*. Multiple agents can exist, and they can handle different signals of a DUT. For example, in a design with a read interface and a write interface, a different agent for each one can be useful, as it allows to insulate different functionalities of the DUT. Each agent may also have its own *coverage* class to apply coverage structures to its connected DUT pins.

The last steps involve checking the items from each monitor. These items are sent to a *scoreboard* where the engineer has defined a reference model. This reference model is an implementation of the expected behaviour of the design. Analysing the driven items, the model decides which outputs are expected next, and compares its expectations with the monitored items.

The agents and the scoreboard are included in a single class, an *environment*. Then, the environment can be reused in different tests, a *test* class. Each test will share the same environment but will implement its own set of sequences to execute. Thus, the engineer can reuse the structure to interact with the DUT and can focus specifically on designing the sequences.

The UVM testbenches allow for all the explained SystemVerilog structures; random constrained stimuli, assertions, and coverage. Thus, a UVM testbench can be considered a significant powerful tool when it comes to functional verification.

Chapter 3. Study of cocotb

3.1 Overview

Cocotb (8) (from coroutine-based cosimulation testbench) is a Python-based framework for the verification of VHDL and Verilog designs. It was first released on July 9th, 2013, as an open-source project from Potential Ventures, supported by Solarflare Communications Ltd (9). Since mid-2018, the project has been administrated by the *Free and Open-Source Silicon* UK-based, non-for-profit, foundation (FOSSi). The source code is hosted on a GitHub repository (21) protected under the BSD-3-Clause license, which allows its usage and modification by any user without any cost, including industrial environments.

In essence, cocotb allows its users to perform the verification of HDL designs running testbenches written in the Python programming language. The framework works with many existing simulators, both proprietary and open-source ones (22).

The project is updated regularly; four new minor versions were released in 2021, four in 2022, two in 2023, and there is already a new major version in the works. It is sponsored by *Cadence Design Systems*, *Siemens Digital Industries Software*, and *Aldec* (8). These EDA tool providers supply simulator licenses to cocotb developers for integration testing.

Additionally, cocotb was also part of the Open-Source Silicon Conference (ORConf) of 2023 (10). This backed-up support by significant and popular EDA companies, as well as participation in industry-focused conferences, has made cocotb known across the verification landscape. During the ORConf conference, information from a Siemens research study showed that cocotb is an alternative being used as a functional verification tool for ASIC and Field-Programmable Gate Array (FPGA) testbenches in real design projects.

One of the most significant cases of industrial adoption of cocotb is presented by the European Council for Nuclear Research (CERN) on its ATLAS experiment, in collaboration with the University of Pennsylvania (23). In the experiment, two ASICs were successfully verified using the cocotb framework, and the verification team is planning to use the same approach in the future. Note that in this case, the plain cocotb library was used with no UVM implementation.

3.2 Internal functioning

An HDL simulator is a software program that models the behaviour of a design described in languages like SystemVerilog, VHDL, or similar. It reads the provided file, compiles the human-readable code into a connectivity description between logic components (a *netlist*), and elaborates a model of how the design works. As such, a simulator transforms HDL design code into computer operations to replicate its behaviour.

Behavioural simulators do not change the input signals of a design on their own, a stimuli generator is needed; a testbench. The testbench is also written and described in HDL code, thus it can also be interpreted directly by the simulator. Nonetheless, simulators allow for the execution of already compiled C/C++ code. Through dedicated interfaces, the C/C++ programs are granted access to the data contained in a simulation, providing the ability to actively change the state of a design. This code can either be started right away or be called from an HDL testbench; an example of the latter scenario is shown in Figure 5.

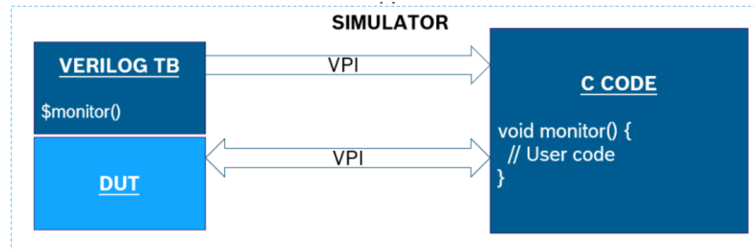

Figure 5. Block diagram example of C code execution in a simulator using VPI.

Each HDL describes its interface, thus resulting in the existence of multiple interfaces such as the *VHDL Procedural Interface* (VHPI), the *SystemVerilog Procedural Interface* (VPI), and the *Foreign Language Interface* (FLI). The latter is a non-standard proprietary interface used by the *QuestaSim* simulator.

At its highest description level, cocotb provides a wrapper for Python to interact with the simulator through the VHPI, VPI, and FLI interfaces using a background C/C++ structure. Nonetheless, its functioning presents an interesting case.

Firstly, note that each HDL uses its own interface. To get around multiple functions, cocotb implements a single Generic Procedural Interface (GPI) in C++ that can effectively communicate the same desired action to the simulator using any of the three mentioned interfaces [(24)].

Subsequently, a problem is met. The simulator is the one that triggers the communication with C/C++ programs, thus a Python testbench cannot directly start the simulation. The solution lies in the Python/C application programming interface (API). This API allows the Python interpreter to be embedded into C/C++ modules, and for extension modules so Python can make use of C/C++ code [(25)].

When cocotb is started, the GPI C++ library is compiled by the framework into a shared library and loaded into the simulator. At the same time, the Python interpreter is embedded into the shared library as well. When the simulation begins, the simulator triggers the Python environment through the embedded interpreter, which runs the Python testbench [(24) (26)].

At this point, the testbench is being interpreted inside the simulator. Thus, when a call to the simulator is made (e.g., changing a signal value), the previously compiled GPI module can be used. The Python directive is communicated to the simulator through the GPI C++ extension module, and the action is performed [(26)].

As a non-developer user, the internals of cocotb may appear complex. Nevertheless, the expected ordinary use does not require to understand these intricacies. Thus, cocotb can simply be seen as a wrapper that allows simulation control from Python. For a block diagram description of how cocotb operates, refer to Figure 6.

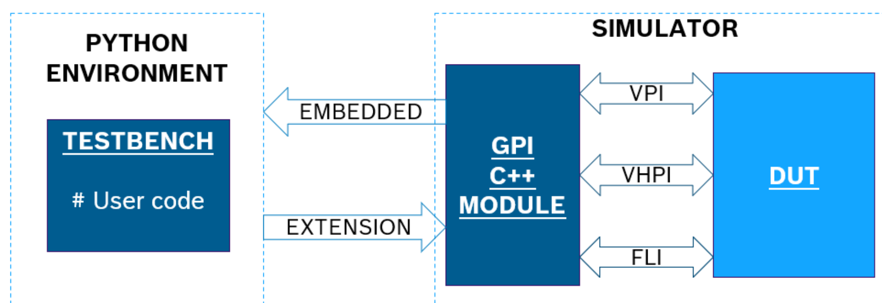

Figure 6. Block diagram on the internal functioning of cocotb.

3.3 Coroutines

In the context of a programming language, a subroutine can be described as a set of statements stored in memory that can be executed externally. A program might call a subroutine, wait until its completion, and then return to the caller execution flow. This behaviour allows for code reusability across a main program and can be easily exemplified with a Python function calling another function.

Coroutines, however, allow for concurrency. While a coroutine is waiting for an event to take place, the execution flow is returned to the caller function, thus freeing up resources for other actions to be executed while waiting. After the event is triggered, cocotb resumes the coroutine execution where it was previously suspended. To do so, cocotb uses a "scheduler". This structure can be seen as an execution flow manager that dictates the order in which different actions may take place.

The cocotb framework is based on the use of coroutines for advancing simulation time; like the SystemVerilog task construct. When a coroutine is waiting for an event to happen (for example, a rising edge of the clock signal), other coroutines can be executed. When the scheduler detects that all the coroutines are waiting for their events, the simulation time is advanced until one of these events is triggered. Then, the coroutines are resumed and the execution continues.

3.4 Basic concepts

The following section makes use of available documentation [27] [28] to explain the basic notions needed to code a testbench using cocotb.

In the cocotb framework, the main entry point of code execution is a *test*. All functions decorated with *cocotb.test()* represent structures where calls will be made to the simulator. These functions can take multiple input arguments, but the documentation encourages the user to only use one; a handle to the DUT. A simple test can be defined as follows:

```
@cocotb.test()
async def tst_base(dut:HierarchyObject) -> None:
    ...
```

A testbench file can contain multiple test functions, each function's name is considered the test's name. Note that the decorated function must be a coroutine, defined by the *async* keyword. Additionally, note that type-hinting the input and output of the function is an optional characteristic of Python.

The DUT object contains the handle for every modifiable object inside it. These objects can be accessed with the standard dot (".") notation. The value of a signal can be changed as follows:

```
dut.clk.value = 0
dut.clk.setimmediatevalue(0)
```

The previous approaches to changing the value of a signal are slightly different. The former schedules the value to be changed at the end of the current simulator step. The latter changes the value immediately. Meta values can also be assigned using dedicated type structures:

```
dut.clk.value = Logic("X")
dut.wr_data.value = LogicArray("01XZ")
```

When reading a signal, the syntax follows conventional rules, with a notable difference:

```
rd_val = dut.rd_data.value
rd_val = dut.rd_data
```

The first statement reads the value of a signal while the second one retrieves the handle of a signal; a reference. The value of a signal is expressed as an integer by default. Nonetheless, an exception will raise if the signal contains a meta value. To avoid this situation, the *value* object provides the *binstr* property that returns the signal value as a binary string.

Bit slicing is also allowed either when writing or reading a signal, as shown below:

```
rd_bit = dut.rd_data[0].value
dut.wr_data[0].value = 0
```

In cocotb, time is advanced through triggers, as explained in the coroutine subsection. When calling these triggers, the Python testbench yields control to the simulator, which in turn advances the simulation time until the trigger is fired. Afterwards, the code continues its execution. The most significant ones are:

```
await FallingEdge(dut.clk)
await RisingEdge(dut.clk)
await Edge(dut.clk)
await ClockCycles(dut.clk, 1, rising=False)
await Timer(1, units='ns')
```

The first two triggers wait for a falling and rising edge of a given signal, respectively. The third one waits until a signal changes its value. The *ClockCycles* trigger waits for n number of cycles of a signal to be completed; the *rising* keyword argument establishes if it should count rising or falling edges. The last one is a timer that waits for a specific time.

Additionally, further structures allow for the combination of multiple triggers, for example:

```
await First(FallingEdge(dut.clk), RisingEdge(dut.clk))
await Combine(FallingEdge(dut.clk), RisingEdge(dut.clk))
```

The *First* trigger waits until one of the specified triggers is completed. The *Combine* structure waits until all the triggers are completed. These types of structures, found in SystemVerilog, are of significant relevance when designing testbenches.

A testbench is not only meant to drive stimuli, but it must also compare the output signals against a designated reference. The *assert* built-in Python statement can be used to trigger an exception and finish the test if a condition is not met:

```
assert dut.rd_data == 20, "Error message"
```

The assertion approach can be useful for errors that are considered fatal; if the assertion condition is not met, the simulation will stop executing the current test and start the next one. A less aggressive alternative is to use the logging capabilities of cocotb.

The DUT object contains a logger object from the Python built-in *logging* library. It allows to print messages of different priority levels during the simulation. The priority level can be changed, hiding messages below the specified level.

```
dut._log.debug("DEBUG")
dut._log.info("INFO")
dut._log.warn("WARNING")
dut._log.error("ERROR")
dut._log.critical("CRITICAL")
dut._log.setLevel(logging.INFO)
```

Note that the *print* Python function does not work during the execution of the simulation, as the console output is managed by the simulator. These *print* statements are buffered and will be shown once the simulator flushes the output buffer.

To run the simulation, a Makefile is used. Some parameters of the testbench to run are needed, such as the simulator to use, the HDL files of the design, the top module of the RTL code as well as the name of the Python file that contains the tests. Additional arguments for the simulator's various stages can also be provided. An example of a basic Makefile is shown:

```
# Simulator options
SIM = xcelium
TOPLEVEL_LANG = verilog

# Design files
VERILOG_SOURCES += $(PWD)/design_1.v
VERILOG_SOURCES += $(PWD)/design_1_a.sv
VHDL_SOURCES    += $(PWD)/design_1_b.vhd

# Simulator arguments
COMPILE_ARGS    = -v93 -sv
RUN_ARGS        = -q -gui

# Testbench description
TOPLEVEL_LANG   = verilog
TOPLEVEL        = design_1     # Design top file
MODULE          = python_tb    # Python testbench file name
```

Note that more options can be specified in the Makefile for a more customizable execution [29].

Once the simulation starts, running the Makefile, each of the tests will run until its completion, or until an assertion error is triggered. Once every test has finished, the simulation will end, and information on the runtime and status of each test will be displayed in the console output.

3.5 PyUVM

Although the cocotb framework can be useful for rapid testing, a main problem arises when it comes to industrial adoption. The *Siemens* research study from 2022 [5] showed that, for ASIC verification, the most used methodology was UVM. Additionally, the study provides data from 2014 and 2018, where this methodology was also in the lead.

The majority within the verification industry uses the UVM approach to verification, thus for an opportunity for Python-based testbenches to find their way into industrial adoption, a UVM library based upon cocotb may simplify the transition.

In this context, two open-source libraries provide a UVM framework using cocotb as their background engine: PyUVM [11] and *uvm-python* [30]. The former introduces a more Python-based approach whereas the latter can be considered a 1:1 port of SV/UVM made for users who have already knowledge of the framework using SystemVerilog. An example of this can be found in the use of the UVM "phase" object, which is not required in PyUVM. Additionally, uvm-python provides a Perl script to convert SV/UVM code into Python. Both are licensed under the Apache-2.0 license, which also allows for its free usage and implementation in industrial environments.

Although uvm-python may appear the best option for this project, it is also important to note that PyUVM presents a more promising future perspective. The latter has had more updates and community activity to its repository in recent years than uvm-python. Furthermore, the author has also published a book on how to use the cocotb and PyUVM libraries [31]. It is also significant to note that PyUVM has received support from *Siemens*, featuring a series of online blogs dedicated to its usage [32].

For these reasons, the availability of resources, community activity, and sponsorship of *Siemens*, this project will use the PyUVM library.

The PyUVM testbenches are structured as the SV/UVM ones. Its reference guide provides enough information on how to set up the verification environment, how to connect the components, and how to run the simulation, which makes use of the same type of Makefile as cocotb.

Nevertheless, the user must have some knowledge of cocotb. SV/UVM connects its agents to the DUT through an *interface* struct, which is a SystemVerilog construct, not a UVM one. Python does not provide an equivalent object, so the user must code an additional object: a *Bus Functional Model* (BFM).

By its simplest description, the BFM will connect the DUT to the driver and the monitor. It must be able to receive sequence items and translate them into DUT signals, and monitor the DUT signals and send a sequence item containing them. To perform these operations, the BFM will use cocotb triggers such as *RisingEdge* or *FallingEdge* to time when to drive and when to monitor signals. Additionally, signals will be written and read which also requires cocotb knowledge.

The following code snippets are shown to demonstrate the simplicity of PyUVM for the creation of an agent and its components, except the coverage class for directness. Note that this is a minimal implementation of a basic UVM agent. Additionally, the inherited classes are imported from the PyUVM library.

This sequence item has two transactional variables. When the item is created it can optionally receive a value for each, defaulting to 0 if not found. An additional function is written, which randomizes the value of one variable.

```python
class MySeqItem(uvm_sequence_item):
    def __init__(self, name:str="m_item", **kwargs) -> None:
        super().__init__(name)

        self.i_wr_data  = kwargs.get("i_wr_data", 0)
        self.o_rd_data  = kwargs.get("o_rd_data", 0)

    def randomize(self) -> None:
        self.i_wr_data  = randint(0, pow(2,32)-1)
```

The next object is the BFM. It provides a queue of a single space for the driving items and an infinite queue for the monitored items. It also has direct access to the DUT pins. Two functions are dedicated to handle the interaction with the queues, and an additional function resets the input signal. Finally, two background functions are initiated when *start* is called. The *_driver* structure waits for the next rising edge of the clock signal and waits until there is an item in the driver queue; once the item is found the input signal is relayed to the DUT. The *_monitor* function on the other hand waits for the falling edge of the clock signal and then creates a sequence item in which it stores the signal values from the DUT; it then stores the item in the monitor queue.

```
class MyBfm():
    def __init__(self) -> None:
        self.dut        = cocotb.top
        self._driver_q  = Queue(maxsize=1)
        self._monitor_q = Queue(maxsize=0)

    async def get_item(self) -> MySeqItem:
        seq_item = await self._monitor_q.get()
        return seq_item

    async def send_item(self, m_item:MySeqItem) -> None:
        await self._driver_q.put(m_item)

    async def reset(self) -> None:
        self.dut.i_wr_data.setimmediatevalue(0)
        await RisingEdge(self.dut.clk)

    async def _driver(self) -> None:
        while True:
            await RisingEdge(self.dut.clk)
            seq_item : MySeqItem = await self._driver_q.get()

            self.dut.i_wr_data.value = seq_item.i_wr_data

    async def _monitor(self) -> None:
        while True:
            await FallingEdge(self.dut.clk)
            seq_item = MySeqItem("seq_item")

            seq_item.i_wr_data = self.dut.i_wr_data.value
            seq_item.i_rd_data = self.dut.i_rd_data.value
            self._monitor_q.put_nowait(seq_item)

    def start(self) -> None:
        start_soon(self._driver())
        start_soon(self._monitor())
```

The following snippets refer to the monitor and the driver components. Each of them retrieves the BFM object from a shared database. The monitor additionally creates a port to communicate with other components.

The driver object applies the reset of the BFM and starts its background functions. Afterwards, it waits until the sequencer sends a new sequence item and then forwards it to the BFM driver queue. The monitor waits until there is a new monitored item from the BFM and then sends it through its port.

```
class MyDriver(uvm_driver):
    def build_phase(self):
        self.bfm = ConfigDB().get(None, "", "my_bfm")

    async def run_phase(self):
        await self.bfm.reset()
        self.bfm.start()
        while True:
            seq_item = await self.seq_item_port.get_next_item()
            await self.bfm.send_item(seq_item)
            self.seq_item_port.item_done()

class MyMonitor(uvm_monitor):
    def build_phase(self):
        self.bfm = ConfigDB().get(None, "", "my_bfm")
        self.analysis_port = uvm_analysis_port("analysis_port", self)

    async def run_phase(self):
        while True:
            seq_item = await self.bfm.get_item()
            self.analysis_port.write(seq_item)
```

Finally, the agent class creates the driver, the monitor, a sequencer, and a port. Subsequently, the driver is connected to sequencer and the monitor port is connected to the agent port.

```
class MyAgent(uvm_agent):
    def build_phase(self):
        super().build_phase()
        self.m_seqr      = uvm_sequencer("m_seqr", self)
        self.m_driver    = MyDriver("m_driver", self)
        self.m_monitor   = MyMonitor("m_monitor", self)
        self.a_port      = uvm_analysis_port("a_port", self)

    def connect_phase(self):
        super().connect_phase()
        self.m_driver.seq_item_port.connect(self.m_seqr.seq_item_export)
        self.m_monitor.analysis_port.connect(self.a_port)
```

The following code represents a simple sequence that the agent could execute: a sequence item is created and randomized.

```
class SimpleSeq(uvm_sequence):
    async def body(self):
        seq_item = MySeqItem()
        await self.start_item(seq_item)
        seq_item.randomize()
        await self.finish_item(seq_item)
```

To finish the testbench, a top environment and a test is needed. Note that the following environment does not implement a scoreboard, for simplicity of the code snippet. The agent port would have been connected to the scoreboard object, as shown in Figure 4.

The environment contains the designed agent and creates the BFM object. This BFM object is set into a database so that other components can retrieve it and use it. The environment is instantiated into a test, where sequences can be executed (started) using the adequate sequencer.

```
class TopEnv(uvm_env):
    def __init__(self, name:str="top_env", parent:Any=None):
        super().__init__(name, parent)
        self.my_ag : MyAgent = None

        ConfigDB().set(None, "*", "my_bfm", MyBfm())

    def build_phase(self):
        self.my_ag = MyAgent("my_ag", self)

@test()
class TstTest(uvm_test):
    def build_phase(self):
        self.env = TopEnv("m_env", self)

    async def run_phase(self):
        self.raise_objection()
        SimpleSeq.start(self.env.my_ag.m_seqr)
        self.drop_objection()
```

Note that the previous code provides a simple example with missing capabilities such as coverage or a scoreboard. Nonetheless, the reader should be able to understand the basic testbench structure in PyUVM.

Chapter 4. Synchronous First-In First-Out

For the initial usage case, a design of a synchronous first-in first-out (SFIFO) will be used. This kind of circuit is one of the most common digital designs, and its simplicity in functioning makes it ideal for verification training as well as ramp-up projects.

Three different versions of the design have been provided; two of them present faulty behaviour that is meant to be found. This design makes it an ideal starting point to design a testbench in PyUVM for the first time.

4.1 Specifications and verification plan

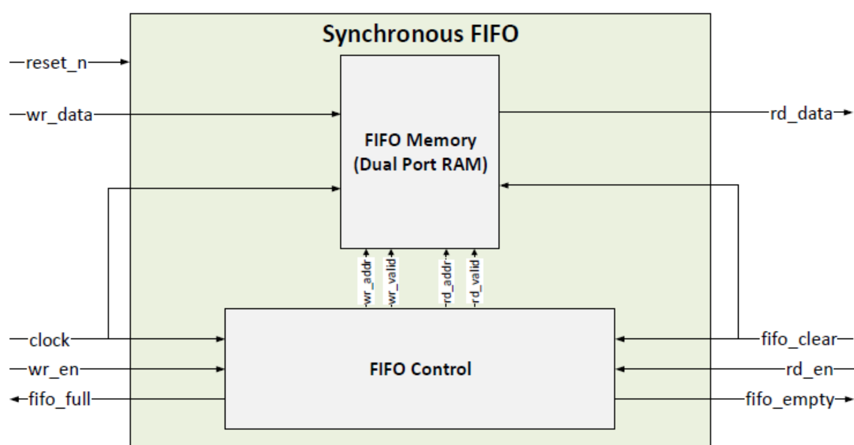

Figure 7. Block diagram of the SFIFO design.

The structure shown in Figure 7 consists of a memory array with user-accessible control logic. New data can be written to the SFIFO and then read in the same order it was written. The main characteristics are as follows:

- Signals are clocked at the rising edge of the clock signal.
- Read and write operations are independent of each other, with their own enabling signal.
- Read data is provided one cycle after the read request.
- Status flags are updated on the following cycle of an operation.
- Read access is ignored if the empty flag is asserted.
- Write access is ignored if the full flag is asserted.
- Output data, memory contents, and status flags can be cleared with the synchronous *clear* signal.
- Reset signal *reset_n* is asynchronous.
- The *clear* and *reset_n* signals have priority over read or write access.

From the previous bullet list, it can be observed that the SFIFO has four different operations; read, write, clear, and reset.

At the same time, although randomization is the key to exhaustive verification, there are specific cases that could be interesting to observe directly without waiting for an unknown number of cycles until randomization produces the needed stimuli. These scenarios are as follows:

- Write access when memory is full.
- Read access when memory is empty.
- Read and write accesses at the same time.
- Reset asserted asynchronously.
- Clear asserted asynchronously.

For these specific cases, the main goal is to observe the behaviour of the design on specific conditions; the test will run until it has completed the action and then it will stop to check that the outputs match expectations. Contrary, the randomized test can run indefinitely until a fault is found.

The completeness of verification is defined via coverage. This can include the SFIFO being full and empty, all possible 8-bit values being written, or other scenarios. To track these insights, the coverage of the following signals will be tracked:

- Read and write data, to obtain a metric on the different values have they taken, with interesting cases for 0 and 255 values.
- Status flags, to check that the SFIFO has been full and empty during the test.
- All combinations of input signals (excluding write data).

For a fully randomized test, it will be considered that the design has been verified when all operations (read, write, clear, and reset) have been performed in all their combinations. Additionally, the SFIFO must be empty and full at least once during the test, and the values 0 and 255 must be written to memory at least once each.

4.2 Reference model

To match the outputs, a model of the SFIFO must be created. Because of its behaviour, an array is the simplest approach, but its implementation varies depending on the programming language.

For the PyUVM testbench, the reference model in Python can use a list to match the functioning of a FIFO structure. Each time there is a new write operation, new data will be added at the end of the list using the *append* method. When reading, the first element of the list will be extracted using the *pop* method. This last method not only gets the value of the indexed element of a list but also deletes it from it. An example follows:

```
# Initial state
model = [0, 1, 2, 3]
# Write operation
wr_data = 4
model.append(wr_data)
print(model)
# Read operation
rd_data = model.pop(0)
print(rd_data)
print(model)
> [0, 1, 2, 3, 4]
> 0
> [1, 2, 3, 4]
```

Note that, by nature, Python lists are not fixed in length and can be expanded as much as wanted.

For the SV/UVM approach, a *queue* SystemVerilog object has been used. This type of object consists of a data array that can append and extract values at the end and the start of the array. To correctly model the SFIFO behaviour, written values will be pushed at the end of the queue, while read values will be extracted from the start of the queue. An application example of this type of object follows:

```
// Initial state
logic [7:0] model [$:7];   // A queue of 8 slots of 8 bits each
model.push_back(0);
model.push_back(1);
$display(model);
// Write operation
byte wr_data = 2;
model.push_back(wr_data);
$display(model);
// Read operation
byte rd_data;
rd_data = model.pop_front();
$display(rd_data);
$display(model);

> [0, 1]
> [0, 1, 2]
> 0
> [1, 2]
```

Note that the *$display* function cannot be used directly with objects as shown in the previous code. The purpose nonetheless is to observe the state of the queue after performing the operations.

Note also that both model implementations have control structures over the length of the memory array to replicate the SFIFO status flags.

4.3 Testbench structure

The following section explains the structure followed in designing the UVM testbench to use. Both, the SV/UVM and PyUVM approaches, will be based upon this structure of components and their connections.

For the PyUVM approach, there are two different implementations. One of them generates the clock using the cocotb *clock* class inside the Python file, whereas the other one generates the clock signal in a SystemVerilog auxiliary file. The purpose is to later observe how it affects simulation runtime.

Regardless of the implementation, when designing a UVM-based testbench, deciding the number of agents and their scope is a vital part of the process. There is not a single valid answer to this question, because it mostly depends on the number of functional interfaces needed.

In the case of this device, three main interfaces can be easily identified:

- **Write**: Drives the enable and data signals of the write operation.
- **Read**: Drives the enable signal of the read operation and monitors the read data.
- **Control**: Drives the *reset_n* and *clear* signals and monitors the status flags.

While no output signal is being monitored by the *Write* agent, it is necessary to track the stimuli used so that the scoreboard reference model can match the expected behaviour. Thus, while not monitoring any output, the write agent also incorporates a monitor component.

This agent configuration is meant to separate the interfaces by their operations, which allows for a simple implementation of concurrency. Each operation will have its own dedicated sequence, and because each of them uses a different interface, they can be run simultaneously.

The testbench structure in Figure 8 shows all agents instantiated on the top environment alongside the scoreboard, which contains the reference model.

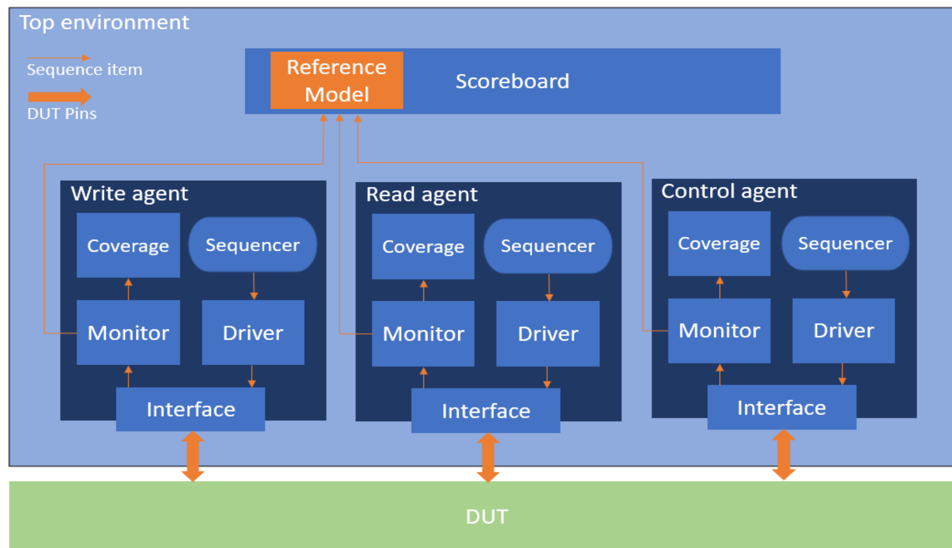

Figure 8. UVM structure of the SFIFO testbench.

The *Write* agent contains three sequences. The first one asserts the enable signal and randomizes the write data, while the second one de-asserts the enable signal. Splitting the write transaction like this gives more control to the verification engineer, who can let the enable signal on a high level indefinitely if needed. Finally, the third sequence randomizes the write data and activates the enable signal with a probability of 70%; this sequence is meant to be used for a high-level randomization test.

The *Read* agent presents the same sequence structure as the one just explained: enable transaction, disable transaction, and enable transaction with a 70% probability. No data is randomized because read data is an output signal of the DUT.

Note that the probability value has been chosen arbitrarily. In the verification plan, it has been explained that the SFIFO should be checked on full and empty scenarios. If for every write access there was a read, the "full" condition could not be met. On the other hand, a probability of 50% for each access can keep the simulation running without reaching either condition. A 70% probability presents the possibility of concatenating multiple accesses of the same type, thus amplifying the chances of reaching a full or empty SFIFO at the same time as allowing other states.

At last, the *Control* agent. It contains a sequence for activating the *reset_n* signal and deactivating it afterwards and the same sequence for the *clear* signal. Additionally, there is also another sequence that enables the reset signal with a 0.05% probability and enables the *clear* signal with a 0.5%, providing the setup for every combination of both signals.

For the last sequence of the control agent, probability values for the *reset_n* and *clear* signals have been decided based on functionality. Constantly clearing or resetting the SFIFO to its initial state decreases the chances of "full" or "almost full" scenarios.

It could be noticed that the write and read accesses are recurrent across tests, and the split between the enabling and the disabling sequences could be tedious. It is repetitive, and where there is repetition there is reusability. There exist two top-level sequences called *WriteCycles* and *ReadCycles* that perform a defined number of read or write accesses, de-asserting the enable signal once the access is finished.

Although the basic components have already been explained, the testbench is missing its final element: the tests. The simulator needs an execution entry point that coordinates which sequence to run.

Following the verification plan goals, the following tests have been designed:

- **TstWriteFull**: Performs a write access per memory slot available, and an additional one. Afterwards, the memory is read completely.
- **TstReadEmpty**: A read access is performed after a reset. Later, a random number of write accesses are performed, followed by the same number of read accesses, and an additional one.
- **TstReadNWrite**: Read and write accesses are performed concurrently for a random number of times.
- **TstReset**: Writes a random number of times and a reset operation is performed afterwards. Subsequently, a random number of write accesses take place and later, the SFIFO is read completely.
- **TstClear**: Mimics the structure of TstReset but instead of applying a reset signal, the clear signal is activated.
- **TstRandom**: The random sequences of each agent are run concurrently for a given number of cycles. This test is meant to be the most exhaustive one, as all the input signals are randomized at the same time, every time. By default, the number of cycles is set to 5000; the time span should be enough for the conditions to be met. If not, a new regression can be run again.

4.4 Test results

There are three different versions of the SFIFO design, two of them contain known errors meant to be found. The following simulations are run using the default configuration (256 8-bit slots). Nonetheless, to observe the waveform of certain errors, a smaller memory length will be used (8 8-bit slots) for the sake of visual ease.

Both the PyUVM and SV/UVM testbenches were simulated, and the observed results matched between approaches. The following analysis of the test results is based on the PyUVM testbench.

First version result:

```
**************************************************************************************
** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**************************************************************************************
** fifo_top.TstWriteFull          FAIL        517.00          0.22        2397.26   **
** fifo_top.TstReadEmpty          FAIL         23.00          0.02        1444.60   **
** fifo_top.TstReadNWrite         PASS         86.00          0.06        1411.43   **
** fifo_top.TstReset              PASS         78.00          0.04        2143.44   **
** fifo_top.TstClear              PASS         78.00          0.04        2031.01   **
** fifo_top.TstRandom             FAIL        159.00          0.12        1320.59   **
**************************************************************************************
** TESTS=6 PASS=3 FAIL=3 SKIP=0               941.01          2.48         379.13   **
**************************************************************************************
```

Figure 9. Simulation results of the first SFIFO version.

As described in Chapter 3, cocotb provides a summary at the end of each simulation, highlighting the duration of each test and whether it has failed or not. In Figure 9 it is possible to observe that half of the designed tests have failed. Locating each assertion error will provide further information.

TstWriteFull has failed because the full flag is expected to be 1, but the observed value is 0. Before ruling this scenario as a design bug, it is necessary to assert confidently that the testbench works correctly.

Figure 10 shows eight write accesses being performed properly, with the write enable signal being activated. According to specifications, the full flag should be asserted on the next clock after the last access (at 20 ns), however, this behaviour is not observed. Therefore, the error is triggered by a fault in the design.

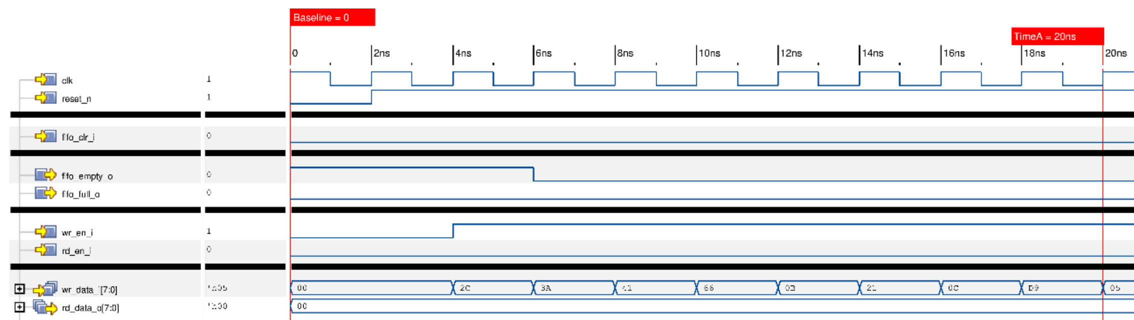

Figure 10. Waveform of TstWriteFull simulation for the first SFIFO version.

On the other hand, *TstReadEmpty* has failed because the empty flag is expected to be one 1 but the observed signal has a value of 0.

In Figure 11 a reset is performed, and a read access follows. Because the memory is empty, the read access does not have any effect on the design. Afterwards, four continuous write accesses take place and the empty flag is de-asserted correctly. To finish the test, four continuous read accesses are subsequently executed.

According to specifications, read data and status flags are not updated until the following cycle after a read access. The first access occurs at 35 ns and no data is given until 37 ns. The last access takes place at 41 ns, and the model expects the empty flag to assert the next clock cycle (43 ns). It is possible to observe that the value of the read data has changed and matches the last written value, thus concluding that the read access works correctly, and the problem lies in the empty flag.

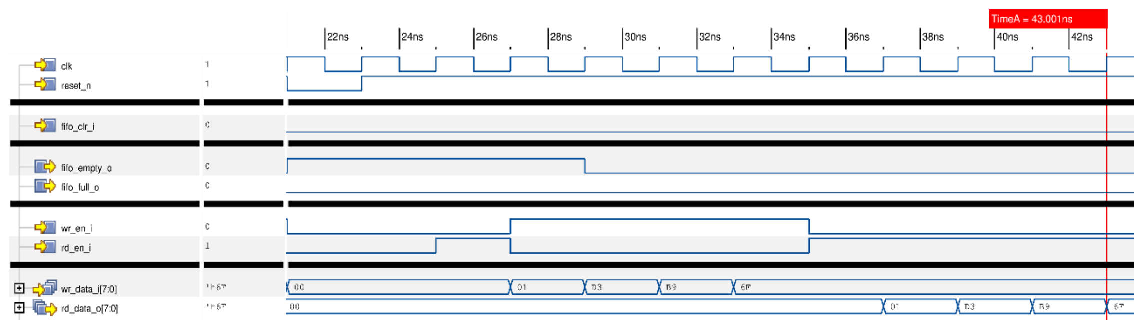

Figure 11. Waveform of TstReadEmpty for the first SFIFO version.

Finally, *TstRandom* has failed because the empty flag is not asserted after reading the entirety of the SFIFO; the same faulty behaviour observed in *TstReadEmpty*.

Second version result:

```
*********************************************************************************************
** TEST                              STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*********************************************************************************************
** fifo_top.TstWriteFull              PASS          40.00           0.03       1352.57  **
** fifo_top.TstReadEmpty              PASS          30.00           0.03       1181.98  **
** fifo_top.TstReadNWrite             PASS          14.00           0.01        967.58  **
** fifo_top.TstReset                  PASS          26.00           0.02       1489.21  **
** fifo_top.TstClear                  PASS          26.00           0.02       1530.18  **
** fifo_top.TstRandom                 FAIL        1319.00           0.94       1396.31  **
*********************************************************************************************
** TESTS=6 PASS=5 FAIL=1 SKIP=0                 1455.01           3.44        423.42  **
*********************************************************************************************
```

Figure 12. Simulation results of the second SFIFO version.

In this run, the second version of the design is the target of the testbench. Looking at the results of each test in Figure 12, it seems like the flag errors have been fixed, although *TstRandom* fails again.

The logging messages of cocotb place the assertion error in a mismatch in the read data signal, which is expected to have a value of 0. The waveform observed in Figure 13 shows multiple read and write access happening, but then the *clear* signal is asserted.

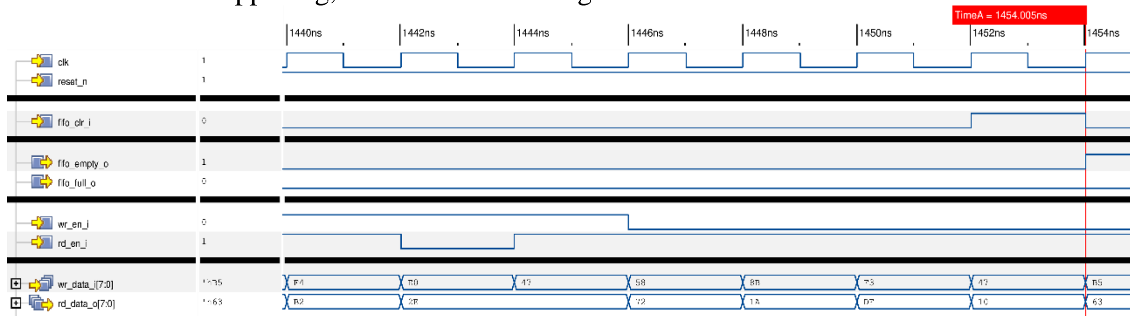

Figure 13. Waveform of TstRandom for the second SFIFO version.

According to specifications, accesses are ignored when the clear signal is asserted, status flags should return to their initial state, and the read data value should be 0. The empty flag is observed to change to 1. Nevertheless, the value of read data is not as expected.

This error has not been observed in *TstClear* because no read operation was performed before the clear; the read data value was always 0. This situation should remind the reader of the importance of randomized stimuli.

Third version result:

```
*********************************************************************************************
** TEST                              STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
*********************************************************************************************
** fifo_top.TstWriteFull              PASS          40.00           0.03       1230.94  **
** fifo_top.TstReadEmpty              PASS          30.00           0.02       1497.62  **
** fifo_top.TstReadNWrite             PASS          14.00           0.01       1146.29  **
** fifo_top.TstReset                  PASS          26.00           0.02       1510.26  **
** fifo_top.TstClear                  PASS          26.00           0.02       1521.51  **
** fifo_top.TstRandom                 PASS       10006.00           7.00       1428.71  **
*********************************************************************************************
** TESTS=6 PASS=6 FAIL=0 SKIP=0                10142.01          10.13       1000.82  **
*********************************************************************************************
```

Figure 14. Simulation results of the third SFIFO version.

For the final design version, every error detected previously has been solved as all tests are seen to pass successfully in Figure 14. Additionally, to assess the validity of the random test, the coverage of each signal can be observed in Figure 15.

Although a significant part of the coverpoints have been completed successfully, the full flag has only taken one of its two values. The waveform reveals that during the random test, the memory has never been full. This situation could be resolved by adjusting the probability values of each type of access, favoring more write access than read, for example. At the same time, tests like *TstWriteFull* have already verified the validity of the full flag.

```
running TstRandom (6/6)
  Random sequence pattern for the SFIFO.
[uvm_test_top.m_env.wr_ag.m_cov]:
WRITE COVERAGE
WR_DATA: 100.0%
WR_DATA_0s: True
WR_DATA_1s: True
[uvm_test_top.m_env.rd_ag.m_cov]:
READ COVERAGE
RD_DATA: 100.0%
[uvm_test_top.m_env.ctrl_ag.m_cov]:
CONTROL COVERAGE
CLEAR: 100.0%
RESET: 100.0%
FULL:  50.0%
EMPTY: 100.0%
TstRandom passed
```

Figure 15. Coverage of TstRandom for the third SFIFO version.

4.4.1 Testbench comparison

The different testbench approaches are compared to provide further insights into their own characteristics. The metrics of the comparison have already been described in the introduction of this document.

The line count metric in Table 1 is based on usable code lines, excluding comments and blank lines. At the same time, both languages allow for multiple ways to write the same statements, varying the line count without any difference in functionality. There has been an attempt to keep consistency in formatting between both approaches.

	SV/UVM	PyUVM (RTL clock)	PyUVM (cocotb clock)
Code lines	1257	870	867

Table 1. Line count comparison for SFIFO testbenches.

Note that the line count for the PyUVM RTL clock implementation accounts for the lines written to generate the clock in the SystemVerilog auxiliary file.

For the runtime comparison, the third version of the design has been chosen because it allows for a longer simulation time since no assertion is going to fail. The test to be run is TstRandom because randomization itself is a resource-exhaustive task from which better insights can be extracted. At the same time, the random test allows for an indefinite number of iterations.

Each approach will execute the test for three different numbers of iterations, providing a detailed observation of how the runtime behaves over time. At the same time, the value of each runtime is an average of 100 executions, to assert that the value is not influenced by computer background activity noise. The GUI mode of the simulator is not being used, and the randomization seed value is stable across all approaches and executions.

Figure 16 shows the results obtained for the runtime comparison, with the values normalized to the slowest simulation time.

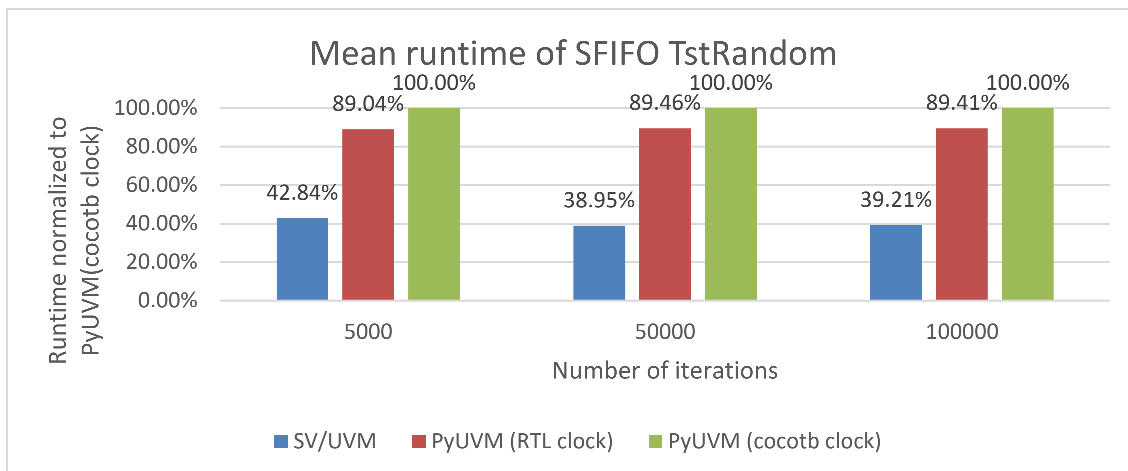

Figure 16. Comparison of the mean runtime of different SFIFO testbench approaches for varying iterations of TstRandom.

4.4.2 Discussion

The artificial design errors introduced in the first versions of the SFIFO have been detected correctly by both the PyUVM and the SV/UVM testbenches. It can be stated thus that PyUVM, and therefore cocotb, are proven working frameworks for the verification of essential digital designs.

The difference in code lines observed in Table 1 is a result of PyUVM presenting less boiler-plate code in its UVM implementation than SystemVerilog. Additionally, Python features such as list comprehension and built-in functions to perform elaborated tasks contribute to a lighter reference model.

Finally, regarding the observed runtime in Figure 16 the SV/UVM implementation is consistently 60% faster than both PyUVM alternatives. However, delegating the clock generation to SystemVerilog instead of cocotb provides a 10% runtime improvement. This improvement, while not comparable to a 60% difference, could also be relevant for prolonged runtimes.

Chapter 5. Bus arbiter

The PyUVM and cocotb frameworks have been evaluated and compared against the SV/UVM approach through the verification of the SFIFO design. Nonetheless, the reasonable next step is to implement a more complex testbench. This section aims to approximate the usage of both frameworks to an industrial application while providing more insight into the comparison between approaches. A bus arbiter will be used as DUT and, once more, two structurally equal testbenches will be designed and simulated; one written in PyUVM and another one written in SV/UVM. However, prior to the design specifications, it may be necessary to provide an overview of how bus arbiters operate.

Usually, multiple devices in an IC may need to communicate with each other. Instead of connecting each block separately with the rest, the idea of a bus comes in handy. It can be summarized as a data line that connects two or more endpoints; a shared wire between multiple devices. Nevertheless, communication on bus lines requires a set of rules.

If all devices are performing accesses to the same target at the same time, no information will be legible; the line will become flooded with data. This set of rules is referred to as a protocol; a common method to share information and to make communication possible. Although significant transmission problems are solved using protocols, there is a functionality missing.

For example, a bus line in a car connects the brake system and the dashboard display (*masters*) with the tire pressure sensors (*slave*). The brake system uses the information to stop the car if a failure happens (one of the tires loses more than 80% of air pressure). The dashboard display just prints the information for the driver to see. If a failure scenario occurs and both masters are trying to access the slave at the same time, safety-wise, access should be granted to the brake system. Priority is needed, and there are protocols that do not enforce it.

Thus, an arbiter acts as the moderator of an “electronic” debate. In case multiple devices try to access the same target, it decides who gets the priority based on specific guidelines. Note that an arbiter is an intermediary that has knowledge of the protocol used and just applies priority rules.

5.1 Specifications and verification plan

The provided design was used as a challenge for verification engineers back in 2018, presenting up to seven known errors to be found. It is considered complex because it requires the implementation of a transmission protocol for the masters and slaves and serves as the characteristic example for agent reusability.

The main basic characteristics of the bus arbiter design can be summed up as follows:

- Signals are clocked on the rising edge of the clock signal.
- Support for three masters (0, 1, 2) and four slaves (0, 1, 2, 3).
- Accessibility to slaves is restricted as shown in Figure 17. Not all slaves are accessible to every master, and the format of valid addresses is restricted in range. Access to a non-valid address does not forward access to any slave.
- Arbitration only takes place if multiple masters try to access the same slave simultaneously.
- Master and slave interfaces implement the AMBA 3 APB protocol.
- A reset signal restarts all accesses.
- The master with the lowest index number has the highest priority.
- A master with lower priority is only granted access when all higher-priority masters are not accessing the same slave or when an access has been completed.
- Slave decoders for master data are combinational.
- The address value selected by a master will be masked in the target slave using the 0x07FFFF mask.

- The bit length of each address is constrained to 24 bits. At the same time, the bit length of the transmitted data is constrained to 32 bits.

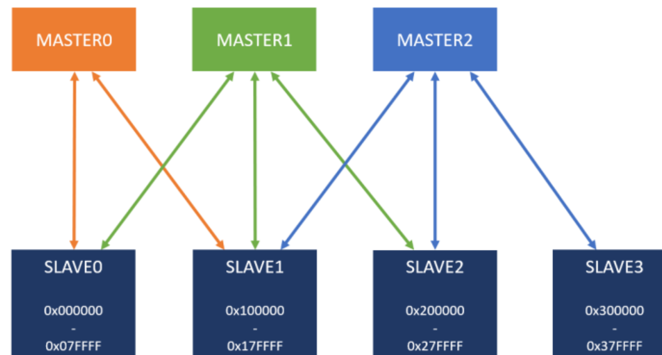

Figure 17. Bus arbiter connection structure and address map.

The arbiter description is unambiguous; however, the implemented protocol (AMBA 3 APB) must be understood prior to the testbench design. Note that the protocol explanation is a simplified version of its real application. Refer to the AMBA 3 APB specifications [(33)] for further information.

Any master can perform two types of accesses to a slave: read and write. To do so, a finite state machine (FSM) is defined by two signals: PSEL and PENABLE (see Figure 18).

- **IDLE**: The initial state where the master performs no operation. Both PSEL and PENABLE are set to 0.
- **SETUP**: The PSEL signal is asserted to 1. In this state, access configuration is prepared. This configuration includes PADDR, PWRITE, and PWDATA (data to be sent) signals, which must remain valid and stable as long as PSEL is set.
- **ACCESS**: The PENABLE signal is asserted to 1. The master is not granted access until the PREADY signal, from the targeted slave, is asserted. After being granted access, the master can either return to the **IDLE** or **SETUP** state.

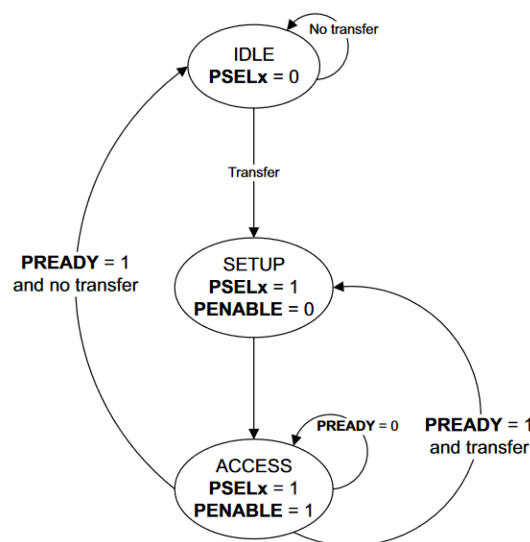

Figure 18. State diagram of the AMBA 3 APB protocol for a master-slave access [(33)].

The following considerations should be noted:

- The PWRITE signal indicates a write (1) or a read (0) access.
- The PREADY signal is only considered valid during the **ACCESS** phase.
- There exists a PSLVERR signal that asserts if the address targeted by the PADDR value is not valid (see Figure 17). Additionally, if this signal is triggered on a slave, it should also be visible on its connected master.
- On a read access type, the data read from a slave (PRDATA) has only to be stable during PREADY being asserted.

With this much information on the functioning of the arbiter, the design of a verification plan may look overcomplicated. Nonetheless, if the protocol description is ignored to focus on the actual target (the arbiter), the following verification goals can be extracted:

- Priority scheme is respected. Masters with a lower index number are preferred, and lower-priority masters are not given access until higher-priority masters are not accessing the same slave.
- The address value at the slave interface is masked using the 0x07FFFF value.
- Incorrect addresses trigger the PSLVERR signal.
- Masters should be able to access their allowed slaves in their address map.
- For incorrect addresses, access should not be forwarded to any slave.
- Read data and write data values between a master and its targeted slave should match during a valid read or write access, respectively.

To accomplish the previous goals, the required coverage must be established:

- Number of allowed master-slave connections.
- Addresses to an existent slave. This includes values outside the valid range so that the address map is evaluated exhaustively.
- The fact that an access can be either read or write.
- All values that the transmitted data (PWDATA and PRDATA) can take.

If all the previous values are meant to have a coverage of 100% the simulation could easily take multiple days to finish. A more realistic goal must be established:

- All master-slave connections are evaluated, even non-allowed ones. This includes 20% of the addresses inside the valid range and 20% of the addresses outside it.
- There are 2^{19} (524288) valid addresses for each master-slave allowed connection, and that number corresponds only to one type of access (read or write). Both types of access must be assessed for 20% of the valid addresses.
- Values for PWDATA and PRDATA must take 5% of the possible values each for valid accesses of their respective type.
- For each master, 20% of the address values that do not point to any slave must be evaluated.

This approach (20% goal for 2^{19} addresses) provides a notable validity to each single simulation run, without the necessity to check every situation. For the data values, 5% has been chosen as the goal since its absolute value (5% of 2^{32}) represents a significant amount altogether.

5.2 Reference model

The most notable behaviour that the model should match is the arbitration scheme. The arbiter assigns a master-slave connection according to addressing and priority rules already described. Nevertheless, the assignment of a master to a slave can only happen when the latter is not being accessed by any other master, or at the end of an existent access phase (when PREADY asserts).

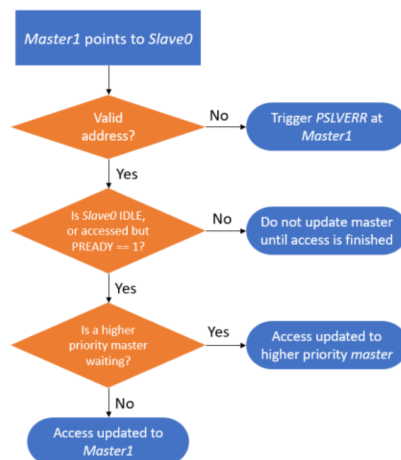

Figure 19. Flowchart for master-slave connection assignment.

Figure 19 describes the decision-making process of the reference model to update a master-slave connection. Addressing a slave with non-valid addresses finishes the process instantly. Additionally, to assign the new master, the targeted slave must be IDLE (not being accessed by any other master) or at the end of an access phase (PREADY=1). Finally, if the slave can be assigned to a new master, the highest priority one is assigned.

This behaviour, while straightforward, presents an added issue. When deciding new connections on time #N, the arbiter checks the value of each master's PADDR signal on #N-1. This means that the model must assign new master-slave connections based on the data of the previous cycle.

Once the arbitration scheme is set up, other checks must be performed. Note that, even when access fails (e.g., non-valid address to an existent slave), PREADY is asserted. In this sense, PREADY indicates that the access phase has finished, correctly or incorrectly.

There are two main scenarios where checks can be performed; prior to an access, and during an access. In the former, master-driven signals must be stable while PENABLE is set. This requires that, every time a master asserts its PENABLE signal, the values of PADDR, PWRITE, and PWDATA are recorded and checked to be the same, every clock cycle until the access is completed. The latter scenario presents a higher complexity.

Because of the nature of combinational logic, found in the slave and master decoders, the assertions cannot always be performed because of signal noise. By specifications, signal matching in a master-slave connection is only considered to take place when PREADY is asserted in the access phase. Thus, checks to compare whether the signals in the master interface match the values observed in the slave, can only happen at the end of a correct access phase.

The first item to model is the PSLVERR signal, which should be 1 at the master interface if it is trying to access a forbidden slave, an incorrect address of a valid slave, or a non-existent slave (according to Figure 17). This situation should also assert that the slave that the master was pointing to cannot have its PSEL signal asserted.

Finally, if the access is valid, then all signals must match between the master and slave, excluding the PADDR signal, which should match the masked value of the master address with 0x07FFFF.

5.3 Testbench structure

The following section explains the structure followed in designing the UVM testbench to use. Both the SV/UVM and PyUVM approaches will be based on this structure of components and their connections.

For the PyUVM approach, there are three different implementations. One of them generates the clock using the cocotb clock class inside the Python file. The second one generates the clock signal in a SystemVerilog auxiliary file. Finally, the third one also generates the clock signal in a SystemVerilog auxiliary file but the BFM of its agents is different.

As explained in Chapter 3, cocotb uses an interface to communicate with the simulator. The usage of this interface is a significant driver of the runtime. To increase performance, one could reduce the calls to said interface by not constantly changing the signal values, for example; instantiating the clock signal on an RTL wrapper.

The improved method of the last implementation consists of creating an auxiliary variable that, for each agent, concatenates the input signals into one, and the same approach for the outputs. In this sense, the PyUVM testbench has been modified so that all driver signals are coded into a single value, sent to the DUT using cocotb, and then decoded on the RTL side. The same applies to the output signal but the other way around. This way, if multiple signals change at once, a single transaction is performed.

Regarding the testbench itself, this design contains an elevated number of input and output signals, and thus separating them into functional interfaces may appear complicated. Nonetheless, the signals can be classified into three groups; related to a master, to a slave, or to arbiter control.

A valid approach could be to control all the signals from all masters using a single agent, but scalability is compromised. It is possible to write a parametric sequence to perform a read/write access by a specific master, however, the problem lies in the sequencer. A sequencer can only manage one item at a time, meaning that multiple sequences cannot run concurrently on the same sequencer. For each master to act independently, it will need an exclusive sequencer. The same logic applies to the slave counterpart.

All masters share functionalities and signal layout; thus, the key is to design a generic agent with basic sequences for a single master, and then create an instance of this agent per each existing master. The agent object is the same, but the interface connected to each one is different. The same logic applies to the slave counterpart. With this approach, the agent object is kept simple allowing for greater scalability and readability.

Subsequently, three different functional interfaces appear:

- **Master**: Drives and monitors the signals of a single master.
- **Slave**: Drives and monitors the signals of a single slave.
- **Control**: Drives the arbiter's reset signal.

In the testbench structure shown in Figure 20, the blocks corresponding to *Master[1,2]* and *Slave[1,2,3]* are simplifications of the *Master0* and *Slave0* agents respectively, and thus share the same component structure as well as sequences.

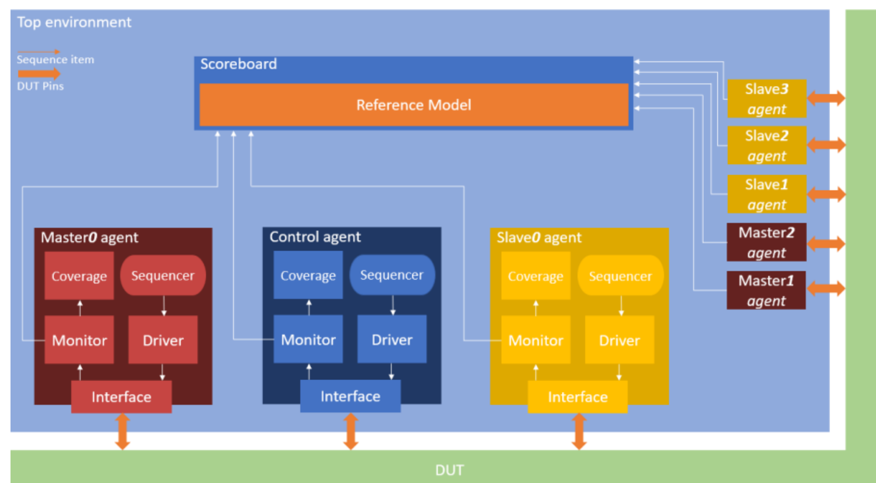

Figure 20. UVM structure of the bus arbiter testbench.

The **control** agent contains two different sequences; one that enables the reset signal and another one to disable it. These two sequences are used separately so that a reset operation of random length in clock cycles can be studied.

The **slave** agents only contain a single sequence; randomize the value of the PRDATA signal. Since a master can read data from a slave, this data should change over time to assert that the observed value at the master interface is valid. Alternatively, additional code structures could have been added so the slave holds the values that other masters have written in the past (during a write access). This would reduce randomization processes (less resource usage for the simulator) while presenting the same validity. Nonetheless, the option to randomize the read value has been chosen to allow for a wider range of values to be evaluated.

The **master** agent is more complex. For starters, its sequence item can be randomized with two distinct functions. Both functions randomize the value of the PWDATA signal, the difference lies in PADDR. The first function needs a target slave so that the address is constrained to point to that slave, including the non-valid addresses. The second function randomizes the address for the rest of the values. These different approaches help to build sequences that cover more cases.

When it comes to the **master** sequences, they are structured following the functioning of the APB protocol. Initially, the master is in an idle state, and it can continue in it or perform a transaction. The first sequence performs a complete transaction; setup and access phase. It has two parameters; whether the access is read or write, and the number of the targeted slave. If the targeted slave exists, it will randomize the PADDR value for the selected slave, else it will randomize the value for the rest of the values. This sequence ends on the access phase so that a new transaction can begin afterwards, but for randomization purposes, the master could also return to its idle state. There is a second sequence that sets the value of PSEL and PENABLE so that the master is in its initial condition.

Coordinating these sequences in a test may be a complex task, although they can be grouped using the same "functional identification" as before. Each master will run independently from the rest, and it will either perform a transaction or stay idle.

In this context, there is a top sequence called **RandomGen**. It requires the number of operations (*n_ops*) to run and the number of the master to use (*n_master*), to select its corresponding sequencer. For each iteration of the loop, the agent must decide whether to stay idle or perform a transaction. For the latter option, it should also randomize the number of the target slave, and the access type (read or write). If the transaction is a read access to an existent slave, the sequencer of said slave is retrieved to run the sequence of PRDATA randomization previously.

The **RandomGen** sequence contains weighted randomization values. For instance, the chance of performing a transaction is 80%, against the 20% probability of staying idle. The idle state does not represent the same probability value as performing a transaction, because the arbitration takes place if there are multiple masters accessing the same slave. Nonetheless, the action of staying idle is a scenario that should also be considered. The loop flow is shown in Figure 21.

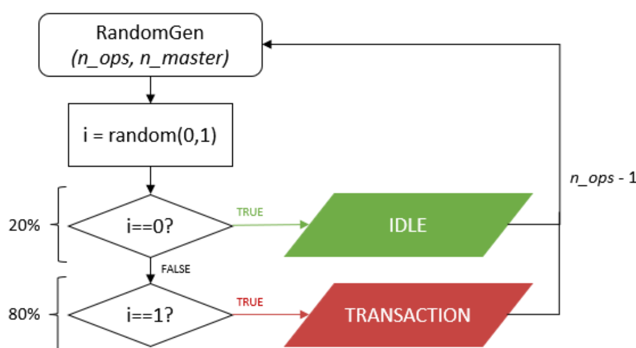

Figure 21. RandomGen sequence flowchart model.

Regarding the transaction sequence itself, because there is no real difference between writing and reading, the type of access is balanced. For deciding the target slave and address, the process is more elaborated.

The purpose of this testbench is to verify the arbiter's functioning, and if the master is mostly pointing to incorrect addresses, then no arbitration is going to happen. From this perspective, a valid slave will be targeted more than half of the time (60% probability), and a valid address will be preferred with a 75% probability. The rest of the time, a master could access a forbidden but existent slave or a non-existent one. The latter represents a wider range of addresses, and if it works correctly and the testbench prioritizes this type of target, most simulation time is going to be spent just asserting that an error should trigger. It has been considered that this scenario should only be checked sporadically with a 10% probability. The remaining 30% is dedicated to an interesting case; a master tries to access a forbidden slave. Multiple results can be obtained from this one, such as validating the address map, or whether the arbiter forwards the access or not.

Figure 22 provides a detailed graph of the transaction sequence and the weighted probability applied for different scenarios.

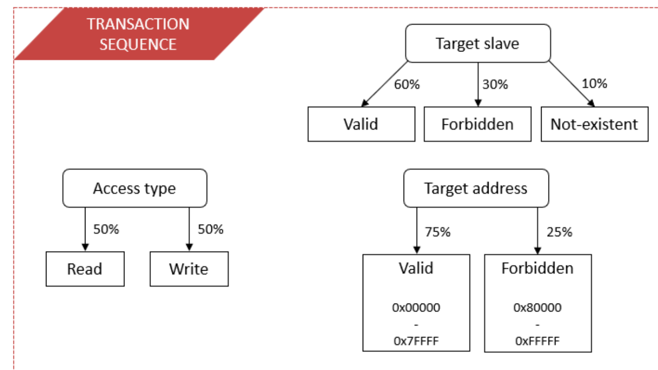

Figure 22. Transaction sequence weighted random stimuli.

There exists another sequence, **NAccess**, that performs *N* continuous read or write accesses (balanced probability) from a selected master to a single specified slave. Afterwards, the master returns to its idle state.

Finally, a **RandomCtrl** sequence is meant to be run for a specific number of operations (*n_ops*) and enable the reset signal with a probability of 0.05%. The reasoning follows the one explained in the SFIFO chapter; multiple resets during a simulation will not provide meaningful results as the number of covered scenarios and chained sequences is low.

These sequences are all meant to be used, although the **NAccess** one has been designed because it provides a specific setup and can be useful to observe an error with a greater level of detail. Nonetheless, because of the complexity of the arbitration process, a single test has been created: **TstRandom**.

This test runs the **RandomGen** sequence for all three masters concurrently between them and, at the same time, the **RandomCtrl** one. The number of operations is set to 5000 by default, considering it an optimal test length when it comes to the number of scenarios to be covered. Nevertheless, the test length can be modified as needed for future simulations.

5.4 Test results

Only one version of the bus arbiter has been provided. As explained in the introduction, this design presents seven known faulty behaviours that will trigger assertion errors. The test will stop once a mismatch between the model and the design is detected, thus cutting the simulation short to observe other errors.

The testbench will be constrained as errors are found. For example, if *Master0* can access *Slave3* and that triggers a model error, then for the next regression, the test will not allow *Master0* to access *Slave3*. This approach assumes that there is just one error between connections, which is specified on the challenge.

Both the PyUVM and SV/UVM testbenches were simulated, and the observed results matched between approaches. The following analysis of the test results is based on the PyUVM testbench.

Error #1: Master0 can access certain Slave2 addresses.

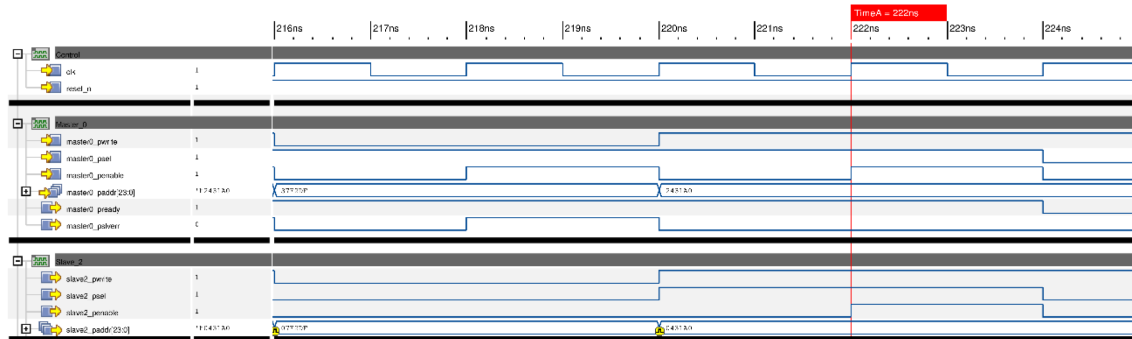

Figure 23. Arbiter error regarding Master0 and Slave2. PSLVERR mismatch.

Figure 23 shows *Master0* trying to access a *Slave3* address at 216 ns, and the PSLVERR signal being correctly asserted at 218 ns. Afterwards, *Master0* tries to perform a write access to a *Slave2* address (0x2431A0) at 220 ns. The bus arbiter address map (see Figure 17) forbids this connection, so PSLVERR is expected to be set at 222 ns. Nevertheless, the signal is never set, wrongly indicating that the access is valid.

Further simulations can help to constrain the scenarios where this same error is triggered. For instance, Figure 24 shows *Master0* trying to perform a write access to a different address of *Slave2* (0x2182BD). This time, the PSLVERR signal asserts as expected.

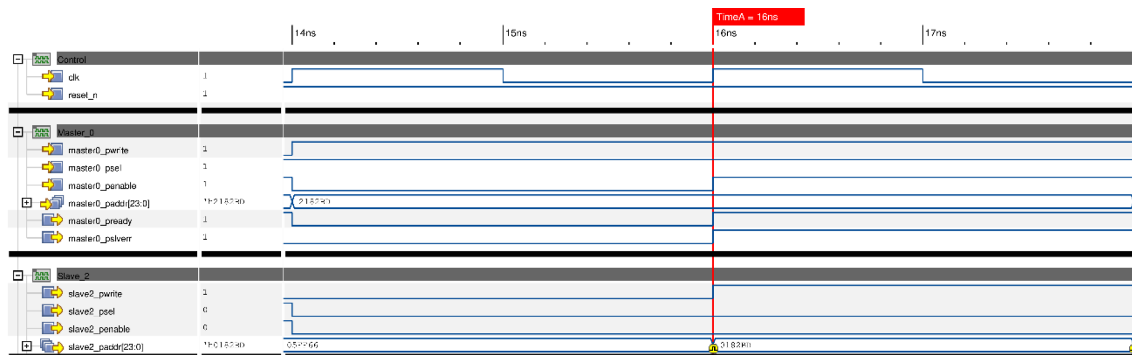

Figure 24. Arbiter error regarding Master0 and Slave2. No PSLVERR mismatch.

If *Master0* cannot access every address of *Slave2*, and if for the same type of access (write) the PSLVERR signal is different, the error must lie on the targeted address of Figure 23**.** Additional simulations have shown the same failure scenario for addresses of the type 0x24XXXX.

Error #2: Slave3 does not mask Master2 addresses.

As per specifications, addresses forwarded to any slave are masked with the hexadecimal value 0x07FFFF. Running the random stimuli test, it is possible to observe a mismatch with this specification description when *Slave3* is accessed from *Master2*. A waveform showing this error can be seen in Figure 25.

It should be noted that it cannot be asserted that *Slave3* does not mask any address. This behaviour is only observed for *Master2* because it is the only master allowed to access. Nevertheless, if other masters were allowed to access *Slave3* it is unknown if the address values would be masked or not.

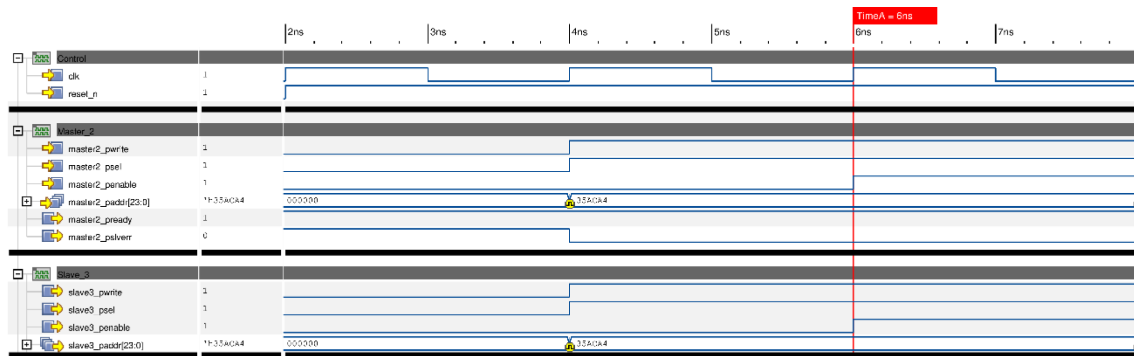

Figure 25. Arbiter error regarding Master2 and Slave3. Unmasked address value.

Error #3: Master2 can access out-of-scope addresses of Slave2.

The waveform in Figure 26, shows signals of all masters as well as *Slave2*. The assertion error is triggered because of a PSEL mismatch at *Slave2*. Observing the waveform, it is possible to assert that neither *Master0* nor *Master1* are accessing addresses to said slave; they are pointing to slaves' numbers 0 and 9, respectively. The only master that is trying to access *Slave2* is *Master2*.

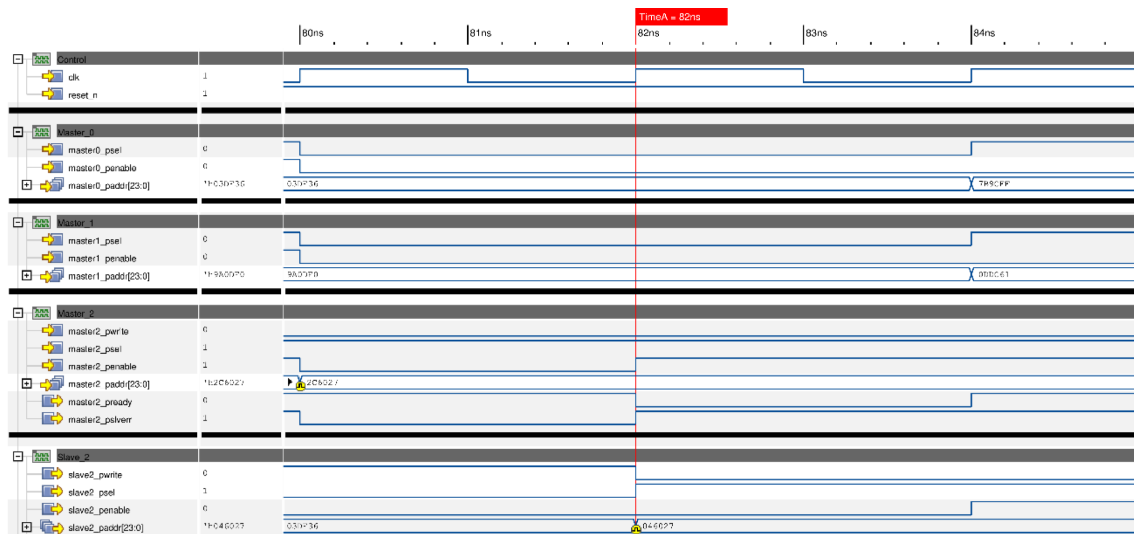

Figure 26. Arbiter error regarding Master2 and Slave2. PSEL mismatch.

The address value of *Master2* is selecting *Slave2* out of the valid scope (0x2C6027 > 0x27FFFF). Because of this, there are no masters correctly accessing *Slave2*, thus the PSEL signal at the slave is expected to be 0. However, the signal is asserted at 82 ns.

The waveform in Figure 27, shows *Master0* and *Master1* pointing to both to *Slave1*, while *Master2* is trying to access an incorrect address of *Slave2* (0x29E5B9). This time the slave recognizes that the address is not valid, and thus the PSEL signal is not asserted.

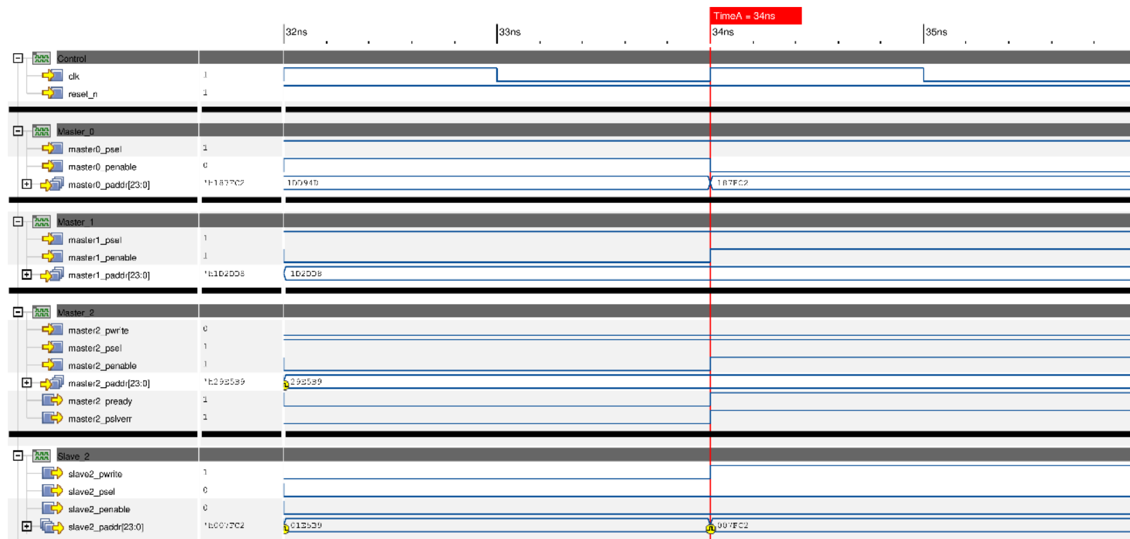

Figure 27. Arbiter error regarding Master2 and Slave2. No PSEL mismatch.

In both Figure 26 and Figure 27, the PSLVERR signal in the *Master2* interface is asserted. The master knows that this access is not valid, but the slave does not react accordingly.

Further iterations of the test fail whenever *Master2* tries to access *Slave2* within the address range of [0x2C0000-0x2CFFFF].

Error #4: Master1 cannot access a valid address for Slave2.

Due to the weighted probabilities established during the testbench structure section, 60% of the accesses are dedicated to a valid slave, and even then, a valid address is selected with a 75% probability. A simple calculation shows that a completely valid access is performed 45% of the time.

This approach presents a natural problem; if there is a short range of valid addresses that fail, the number of iterations to run before detecting them is going to be significant. One could argue that there should be a test to try every possible valid address, but that is computationally not feasible (for the time and resources the simulation will consume). The preferred action plan is to run the test multiple times until the coverage goals are achieved.

In Figure 28 *Master1* is trying to access *Slave2* with a valid address (0x27FFF4). However, the PSLVERR signal rises unexpectedly, failing to meet the expectations of the model.

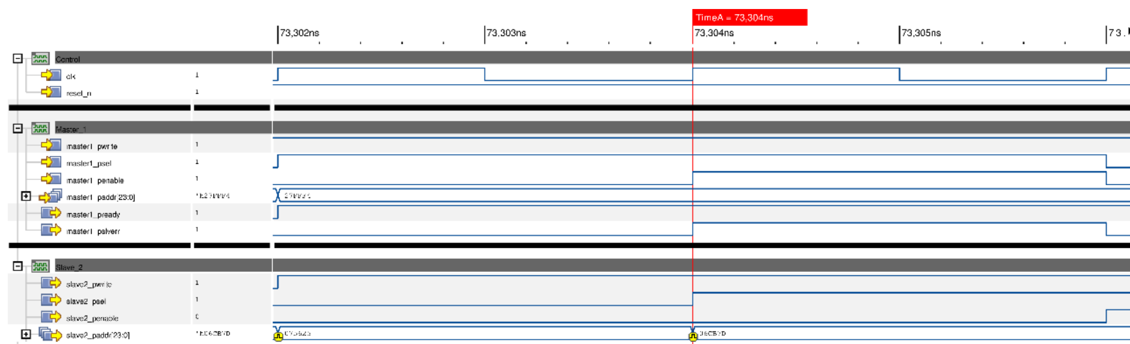

Figure 28. Arbiter error regarding Master1 and Slave2. PSLVERR mismatch.

The same result has also been observed for two more addresses of the kind 0x27FFFX.

In this case, the error is not a convoluted one but straightforward; *Master1* cannot access specific valid addresses for *Slave2*. The problem lies in the conditions needed to find this error.

Evaluating every valid address between *Master1* and *Slave2* would require 2^{19} (524288) accesses. Now, for every master and slave combination, the simulation time required will be notable. Nevertheless, it is accurate and exhaustive. This approach will find the error at the end of the execution. For randomization, a wider range of cases and scenarios are evaluated in the same simulation time, but the probability of pointing to a valid address, in a range of 16 values of a valid slave, is 0.0014 % in the designed testbench.

This error should remind the reader that verification can be addressed from multiple perspectives; prioritize a wider spectrum of cases through randomization, or exhaustively assess specific cases. Both are equally valid approaches, but they serve different purposes and should be used accordingly, depending on the established verification goals.

Error #5: PRDATA mismatch for read access between Master1 and Slave0

In Figure 29 *Master1* is shown trying to perform a read access to a valid *Slave0* address (0x02862C). The targeted slave had been accessed in the past and does not prioritize *Master1* until 212 ns. Then, the access is completed at 214 ns, but an assertion error is triggered by the reference model. A mismatch has been found between the PRDATA value in *Master1* (0xEE51ABA1) and *Slave0* (0x6E51ABA1). The signal is identical except for the most

Figure 29. Arbiter error regarding Master1 and Slave0. PRDATA mismatch (MSB = 1).

significant bit (MSB).

In specific cases during the verification process, errors are not isolated but are the result of a chain of sequences. In Figure 29 it is possible to observe the previous value of PRDATA at *Master1* (0x86FABB7B), which had the MSB asserted.

Comparably, Figure 30 shows a valid read access between *Master1* and *Slave0*. In this case, the access is forwarded at 72 ns and completed at 74 ns. Again, a mismatch on the PRDATA signal is noted: 0x0CD76985 does not match 0x8CD76985. In this case, the MSB should be 1 but it is 0. If the previous value of PRDATA at *Master1* is observed, the MSB is 0.

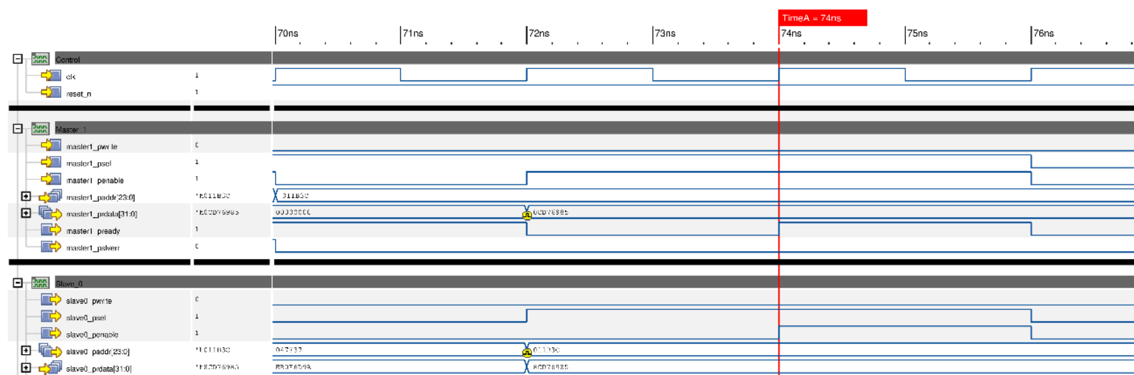

Figure 30. Arbiter error regarding Master1 and Slave0. PRDATA mismatch (MSB = 0).

Further simulations of the test show that when *Master1* performs a read access to *Slave0*, the MSB of the PRDATA signal, at the master interface, maintains its previous value.

Error #6: Masked address mismatch for Master2 and Slave1

Building on the importance of checking chains of sequences, this error may be more representative.

Figure 31 shows *Master2* trying to access *Slave1* using a valid address (0x15115B). The access is delayed because a higher-priority master (*Master1*) is accessing it. The priority is forwarded at 191,584 ns and the access is completed at 191,586 ns. Subsequently, the reference model triggers an assertion error because the masked address observed in the slave interface (0x05114B) does not match expectations (0x05115B).

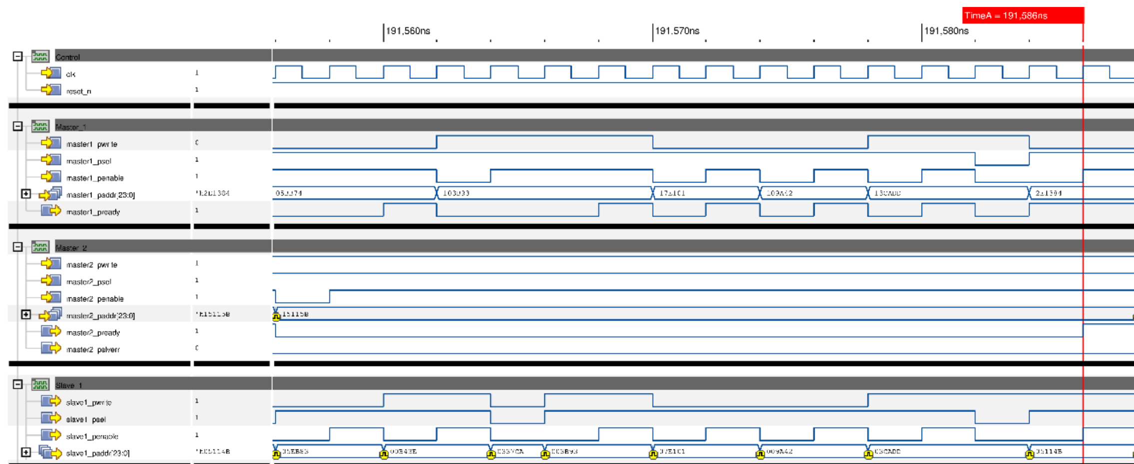

Figure 31. Arbiter error regarding Master2 and Slave1. PADDR mismatch.

This error may be reminiscent of the previous one. In this case, the 5th bit by the right is set to 0 when it should be asserted. The previous value of PADDR (0x03CADD) at *Slave1* had the same bit set, so the same explanation as before may not be valid. To better assess the cause of the error, another regression must take place.

In Figure 32, *Master2* is pointing to a valid *Slave1* address, and the PREADY signal grants access at 54,642 ns after a delay, because of higher-priority accesses. The same error arises; the 5th bit of the masked PADDR value in *Master2* (0x03CC7F) does not match the one observed in *Slave1* (0x03CC6F). Again, the 5th bit value has been set to 0 when it should be asserted.

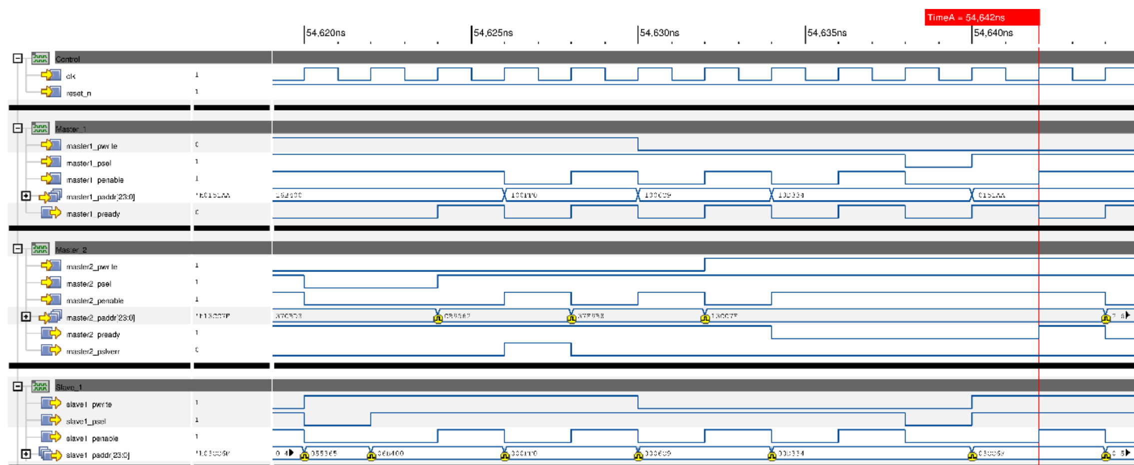

Figure 32. Arbiter error regarding Master2 and Slave1. An additional PADDR mismatch.

Observing the previous figures, the only common trait between them is that *Master2* had to wait because another master had access. The master that blocked the access is the same (*Master1*), but even more of a coincidence is that the number of accesses performed between *Master1* and *Slave1*, prior to the observed error; is the same (4).

Multiple simulations were run, and the error was triggered in the same scenario with the same result. When *Slave1* has been accessed four continuous times by *Master1*, the masked address of a subsequent *Master2* access has the 5th bit (from the right) of PADDR set to 0.

Error #7: PSLVERR not asserting for Master0 and Slave1 (CHECK WAVEFORMS)

In the case of this arbiter, the specifications established that a PSLVERR on a slave should propagate to the accessing master. The errors seen in the master interfaces so far are triggered by wrong addressing, which is managed by the arbiter. A slave can also manage its error signal by itself, presenting an input to evaluate.

The driver component of the slave agents has been modified to assert the PSLVERR signal if the masked address is not 4-byte aligned. This signal assertion should propagate to the accessing master. Note that the specifications do not model this behaviour (triggering an error if the address is not 4-byte aligned), but it provides a method to evaluate if the propagation of the PSLVERR, from a slave to a master, happens.

Figure 33 shows *Master0* accessing a valid *Slave1* address, but not 4-byte aligned (0x170002 % 4 != 0). The *Slave1*, with its newly implemented driver, asserts the PSLVERR signal but it does not match the PSLVERR signal in *Master0*.

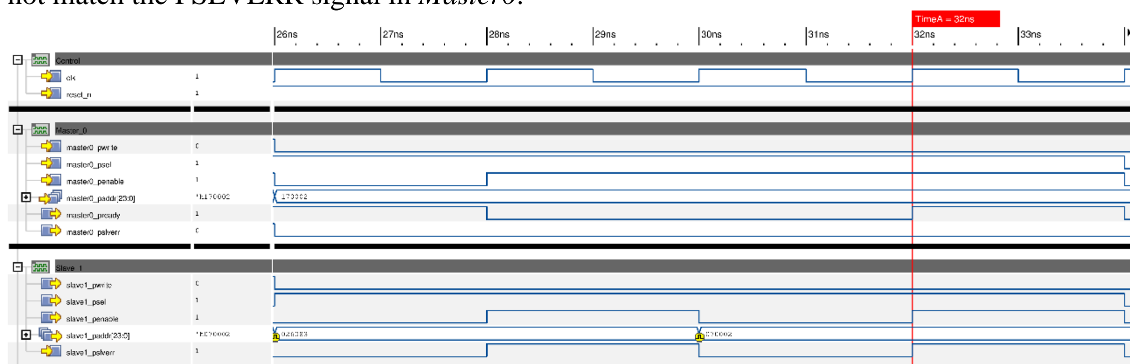

Figure 33. Arbiter error regarding Master0 and Slave1. PSLVERR mismatch.

The error has not been observed in any other master-slave connection. Thus, the faulty behaviour can be described as the PSLVERR signal not being forwarded to *Master0* when it is asserted at *Slave1*.

Obtained coverage.

Although in this case the number of errors to find was known, in a real industrial scenario the end of the verification process can only be asserted through the coverage goals. In this context, the random test has been left to run for a million iterations (each master must perform said number of operations). Both the stimuli and the scoreboard have been constrained so none of the previous errors triggers an assertion error.

The selection of a million iterations is due to previously observed coverage when the test was run to find the known errors. To obtain values close to the 20% specified earlier, a longer simulation must be performed.

```
MASTER1 COVERAGE
SLAVE0 ACCESS: 18.46% valid ; 18.33% non-valid
SLAVE1 ACCESS: 18.42% valid ; 18.44% non-valid
SLAVE2 ACCESS: 16.99% valid ; 19.49% non-valid
SLAVE3 ACCESS: 7.42% valid ; 7.3% non-valid
VALID WRITES:         24.34%
VALID READS:          24.35%
WRITE DATA:           0.00931%
READ DATA:            0.00348%
NON-SLAVE ADDRESSES:  15.22%
```

Figure 34. Coverage of TstRandom for Master1.

To provide an example of the coverage, results associated with *Master1* are shown in Figure 34. As observed, the addresses to allowed slaves (0, 1, 2) are almost covered up to 20%, whereas the accesses to *Slave3* were not as much. Non-slave addresses were also evaluated and read and write operations to valid addresses took place significant times. Nonetheless, the values of written and read data were not near the desired percentage; due to the number of available possibilities.

Coverage should also serve as a metric on how stimuli are generated in the testbench. For instance, the proximity in value between valid write and read accesses, shows that the type of access is an equally weighted random option (50% probability), as described in the testbench structure.

To further increase the coverage, either more simulations can be run, or stimuli generation can be modified.

5.4.1 Testbench comparison

The different testbenches approaches are compared to provide further insights into their own characteristics. The metrics of the comparison have already been described in the introduction of this document.

The line count metric in Table 2 is based on usable code lines, excluding comments and blank lines. At the same time, both languages allow for multiple ways to write the same statements, varying the line count without any difference in functionality. There has been an attempt to keep consistency in formatting between both approaches.

	SV/UVM	PyUVM (enhanced)	PyUVM (RTL clock)	PyUVM (cocotb clock)
Code lines	1548	1069	1033	1030

Table 2. Line count comparison for bus arbiter testbenches.

Note that the line count for the PyUVM RTL clock implementation accounts for the lines written to generate the clock in the SystemVerilog auxiliary file.

For the runtime comparison, the **TstRandom** will be run, because of the resource-exhaustive task that randomization represents. Additionally, applying constraints on the available inputs and which signals to check, the random test allows for an indefinite number of iterations without running into any of the described errors.

Each approach will execute the test for three different numbers of iterations, providing a detailed observation of how the runtime behaves over time. At the same time, the value of each runtime is an average of 100 executions, to assert that the value is not influenced by computer background activity noise. The GUI mode of the simulator is not being used, and the randomization seed value is stable across all approaches and executions.

Figure 35 shows the results obtained for the runtime comparison, with the values normalized to the slowest simulation time.

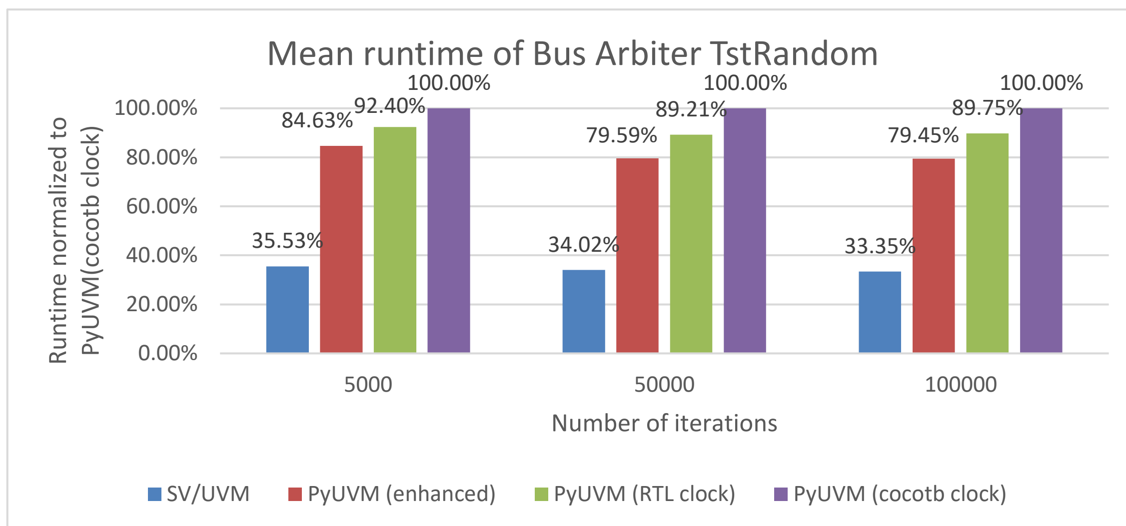

Figure 35. Comparison of the mean runtime of different bus arbiter testbench approaches for varying iterations of TstRandom.

5.4.2 Discussion

The verification challenge has been completed successfully as all the artificial errors have been discovered, by both the PyUVM and the SV/UVM testbenches. It can be stated thus that PyUVM is a suitable framework for the verification of complex digital designs, capable of handling advanced testbench structures.

The difference in code lines observed in Table 2 is a result of an easier and more direct implementation of UVM in PyUVM than in SystemVerilog. Again, the low syntax complexity of Python allows for a reference model that matches the verification goals with fewer statements than its SystemVerilog counterpart. Nevertheless, the enhanced approach of the PyUVM testbench presents more lines of code. This is due to the encoding and decoding structures needed.

Finally, regarding the observed runtime in Figure 35, the PyUVM implementation using the cocotb clock class is the slowest. When the clock generation is described using SystemVerilog instead, the runtime improves by 10%, and using the enhanced approach makes a difference of 20%. Again, the SV/UVM testbench is the fastest, presenting a consistent 65% improvement over the slowest option.

Chapter 6. Chirp Start

Through the bus arbiter example, it has been shown that PyUVM can be used for the verification of complex digital designs. Not only so, but the validity of the designed test has been asserted, as all hidden errors were discovered. At this stage, the relevance of PyUVM as a reliable tool for digital verification may appear significant, however there is a milestone left.

The previously used designs, although complex, were isolated systems with already defined known errors, meaning that they do not represent a completely real verification environment. This chapter will apply every piece of knowledge from the past chapters to a digital design, which belongs to an integrated circuit in active development. The nature of this approach will provide insightful data on how PyUVM operates directly in the semiconductor industry.

6.1 Specifications and verification plan

In ICs, it is a widespread practice to utilize multiple clocks, driven by considerations of efficiency and specific requirements. The use of a higher clock speed inherently provides a superior sample rate. However, not all functional blocks within a system require updates at identical rates. To illustrate this concept further, the analogy of a car can be used again. Most modern cars are equipped with radar sensors to monitor their surroundings. In this context, monitoring other features like air conditioning temperature might not be as critical. Consequently, the radar operates with a higher sample rate, ensuring a faster response to any potential danger.

Nonetheless, the introduction of multiple clocks in a system introduces challenges related to synchronism. Transmitting data between subsystems governed by distinct clocks operating at different speeds demands a level of control to ensure accurate data sampling at the precise moment, preventing information loss. In this context, the Chirp Start block is employed to address what is known as "clock domain crossing", the transfer of data between asynchronous clock domains.

This block connects subsystem A and subsystem B. Each of these subsystems operates on its own internal clock, with one of them commutating at a rate four times faster than the other. The objective is for A to activate B. However, due to the divergence in sampling rates, the duration of the enabling signal from A might be insufficient for B to reliably detect it. This scenario is illustrated in Figure 36, where the START signal is synchronized with *clk*. The pulse width of the START signal, as well as its phase, shows a change in its value outside the *pclk* synchronizer (its rising edge), and thus not being captured.

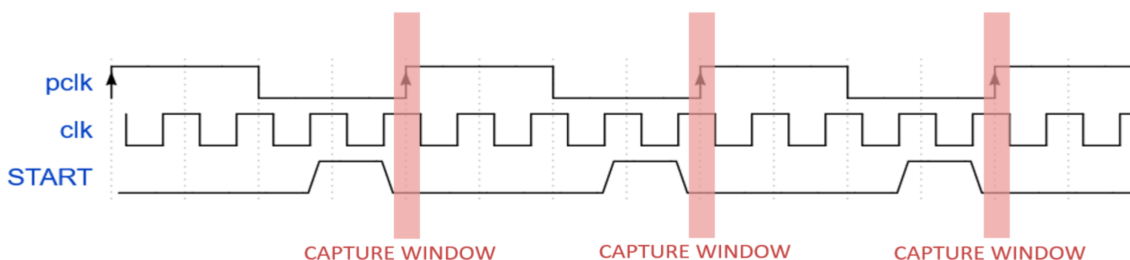

Figure 36. Exemplary waveform of a signal not being captured in a clock domain crossing scenario.

The Chirp Start block is designed to circumvent this situation. Its main objective is to detect the START signal correctly, and then generate it again on the new clock domain. This re-generated signal called RAMP is the same signal as START but translated to the *pclk* domain, as shown in Figure 37.

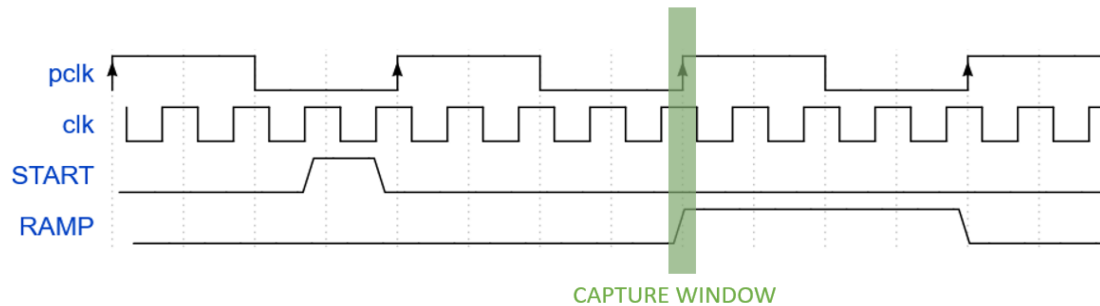

Figure 37. Exemplary waveform of START being correctly translated from the *clk* domain to the *pclk* one.

To achieve this, the Chirp Start design incorporates two distinct functionalities. The first, **calibration**, is responsible for selecting the optimal sampling point to capture START. The second, **relay**, samples START accordingly and then it generates RAMP. A simple block diagram shown in Figure 38 should provide the reader with an overview of the input and output signals, as well as a structural view of the multiple functionalities of this system.

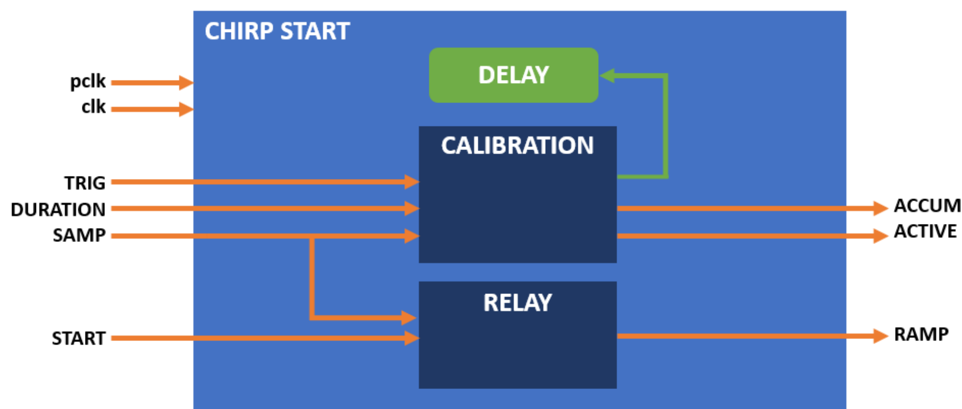

Figure 38. Simplified block diagram of the Chirp Start system.

Note that the modes are mutually exclusive; the system is either being calibrated or acting as a relay.

While the system is enabled, it must receive constantly an external SAMP signal. This signal is a pulse with the width of a *clk* period and must be received once every *pclk* cycle. Its phase relative to the *pclk* rising edge is unknown (can arrive at any point during the *pclk* cycle), but it must be constant on average. An example of this requirement is shown in Figure 39. Note that SAMP sometimes arrives 1 unit before the falling edge of *pclk*, and sometimes 1 unit after it, therefore making the phase constant on average.

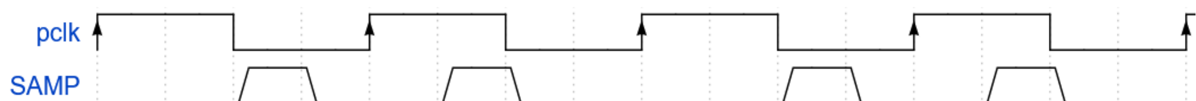

Figure 39. Exemplary waveform on how the SAMP signal must be received.

During calibration mode, the ideal point to sample the START signal is calculated. The method involves the SAMP signal and the *pclk* rising edge.

Every time a *pclk* rising edge is detected, after 2 *clk* cycles the system generates an internal pulse called RS_PCLK. Additionally, every time a SAMP pulse is detected, after DELAY *clk* cycles, the system generates an internal pulse called RS_SAMP. Figure 40 shows a waveform of this scenario.

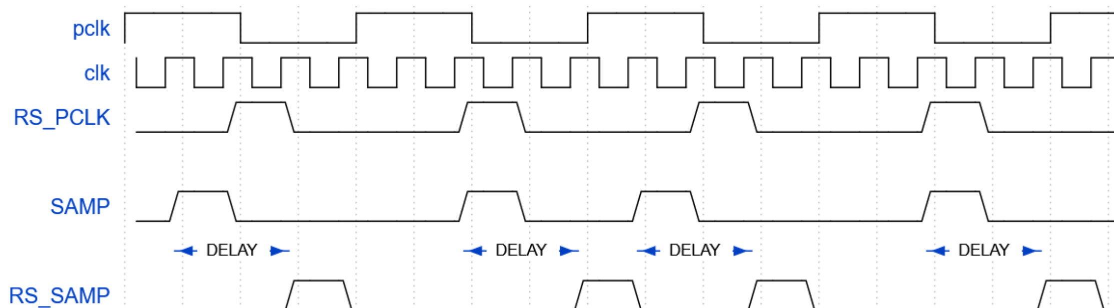

Figure 40. Exemplary waveform on how internal signals RS_SAMP and RS_PCLK are generated.

Note that both pulses have a width of a *clk* cycle, and that both are asserted with a *clk* rising edge. The SAMP signal is received as explained before; although its phase relative to the *pclk* rising edge varies, it is constant on average. The DELAY register has a default initial value.

These internal signals of the system are then used during the calibration mode, to measure the ideal point to sample START. By its most basic description, the system measures the number of *clk* cycles between RS_PCLK and RS_SAMP falling edges. This number is added to ACCUM and, once the calibration is finished, a new DELAY value is calculated by an internal equation. Figure 41 shows the calibration process.

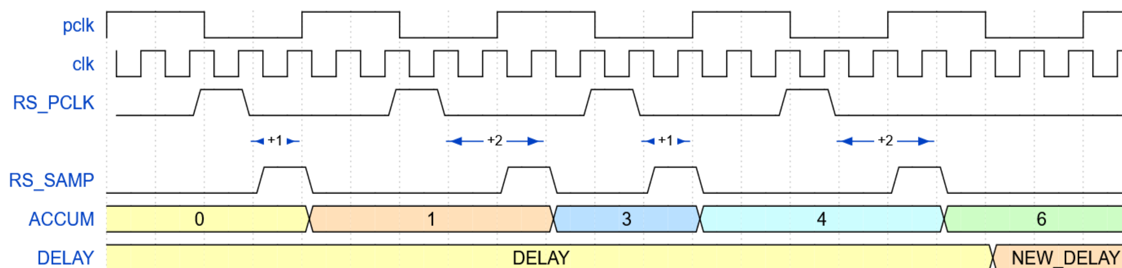

Figure 41. Exemplary waveform on how the calibration mode operates.

Note that, because the SAMP signal has a 1-unit error, the system sometimes counts 2 *clk* cycles and other times counts 3 *clk* cycles; this is the correct behaviour. The new ideal sampling point of START is closely related to the new calculated value of DELAY.

This calibration mode is triggered by a pulse of the TRIG signal. The duration of the calibration is established by the DURATION signal. The calibration starts when the system asserts the ACTIVE signal. This behaviour is shown in Figure 42.

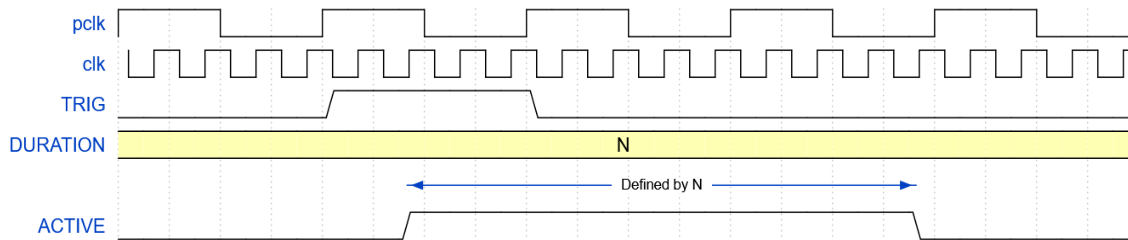

Figure 42. Exemplary waveform on how to trigger the calibration mode.

Once the calibration is done, the ideal point to sample START has been calculated and the relay mode can take place. Once a START pulse is received, the system generates an internal signal RS_START for 1 *pclk* cycle. The value of RS_START is then sampled when RS_SAMP is on a high level. In the next pclk cycle, the captured value is generated in the RAMP output.

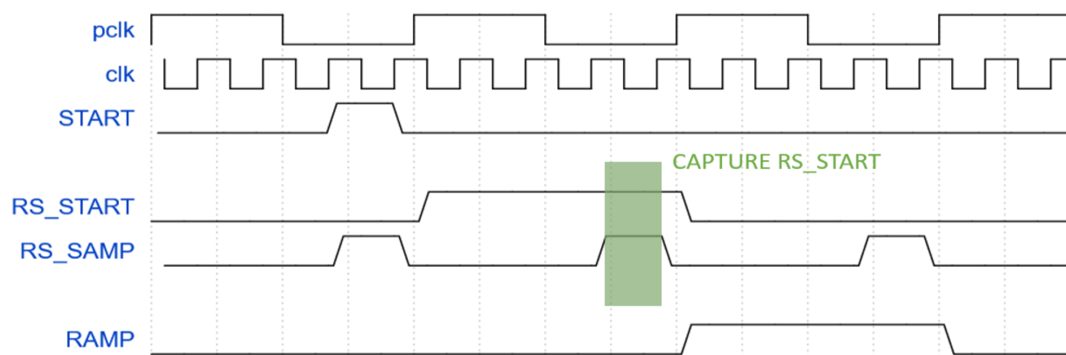

Figure 43. Exemplary waveform on how the relay mode operates.

As shown in Figure 43, the system has successfully translated the START signal from the *clk* domain to the *pclk* domain as RAMP.

While the block description introduces more complexities compared to previous designs, such intricacies are expected in a subsystem embedded within a real IC. Nevertheless, the existence of such distinct functionalities allows for the formulation of a verification plan with distinct objectives in a simplified manner. For the calibration component, the following checks are conducted:

- The ACTIVE signal is asserted for the duration of the calibration.
- The ACUMM value is accurately calculated and retained until the initiation of the next measurement, after which it is reset to 0.
- The internal DELAY value is computed in accordance with the internal equation.

Simultaneously, the relay part must verify that for a given START input, a RAMP output signal occurs as per the depicted behaviour in Figure 37.

Ideally, the testing process should encompass the coverage of DURATION and DELAY values. The former signal evaluates the equation's validity for low-measurement cycles, while the latter ensures that the block functions for every conceivable delay value. This comprehensive approach facilitates a robust verification process for the Chirp Start block.

6.2 Reference model

6.2.1 Clocking issues

Before describing the reference model behaviour, an issue regarding the existence of two independent clocks must be addressed.

For instance, let it be assumed that each signal is sampled by their clock's falling edge. The model could implement two concurrent functions that update, or read, from it whenever a falling edge of any of the clocks takes place. This option works correctly until both falling edges happen at the same time. If the model must be updated before it is read, concurrency allows for a possibility where race conditions may occur. The model could be read before it is updated and subsequently generate assertion errors due to a failure on the model, not of the design itself.

For this reason, to avoid race conditions, all signals will be monitored on the falling edge of the fastest clock.

6.2.2 Model behaviour

During the length of the simulation, the model will generate its own resampled version of the *pclk* edges and the SAMP signal. Both signals are required either for calibration or relay operation, thus their generation is needed.

For the calibration, a measurement takes place whenever the TRIG signal is set to 1, thus for the model to start its internal measurement function it must wait until this signal is asserted. However, the ACTIVE signal is not asserted right away, but once the first resampled SAMP is registered. Additionally, ACTIVE must not be 1 if the block was not enabled in the first place, or if TRIG was not activated.

Once ACTIVE is set, a timer will start and wait for the expected duration of the calibration. During this time, the model will enable a background function to calculate the value of ACUMM using its resampled signals. This value will be compared each time against the monitored value.

Finally, at the end of the measurement, the ACTIVE signal must return to 0 and the new DELAY value must match the value of the equation described in the specifications.

During the relay function, once a START signal is detected, the process described prior in Figure 43 will take place. The model will initiate a background function to assert RAMP at the right moment. The value of the RAMP signal is compared each time against the monitored value.

6.3 Testbench structure

In this chapter, the testbench will only be implemented using the PyUVM framework, to focus all efforts on the verification of the DUT over the comparison of testbench approaches. The PyUVM implementation is designed to efficiently manage the interactions with the simulator, as the PyUVM enhanced approach on the bus arbiter.

This design connects two blocks of different clock speeds together, and at the same time presents two functionalities. The selection of interfaces based on functionality (calibration or signal relay) may be appealing, but each of these functionalities contains signals from both clock domains; complicating the agent's components since each signal would have to be driven and monitored at different clock edges.

In this context, the division of agents based on clock domains appears the simplest, and it does not present any qualitative disadvantage. The only significant modification is that SAMP is a signal that must be generated constantly, and for the sake of simplicity to drive the other signals, SAMP may have its own agent. With this approach, the other *pclk* signals are free to be driven at any time, without potentially being blocked by the generation of SAMP, which agent is relinquished to run in the background.

- **PCLK** agent: Drives every input signal, except SAMP.
- **CLK** agent: Monitors every output signal, and drives RESET_N.
- **SAMP** agent: Manages the generation of the SAMP signal.

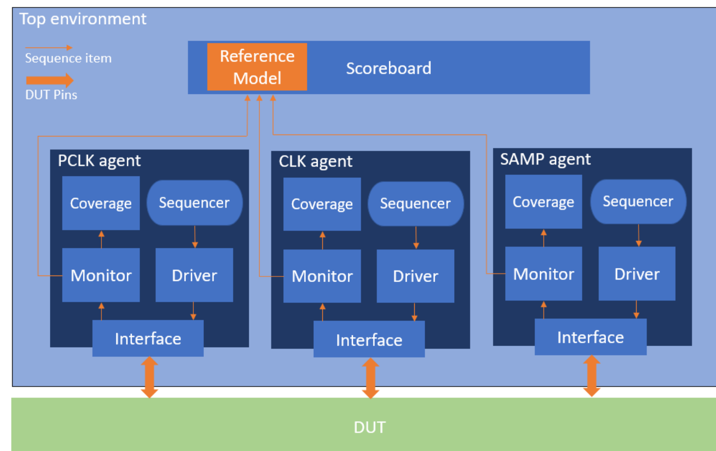

Figure 44. UVM structure of the Chirp Start testbench.

The **PCLK** agent presents a customized driver component for the TRIG signal. Since it must be a pulse, the driver will automatically set it to a low level after it has been asserted. For this, once a calibration trigger is registered, a background function is initiated to wait the necessary time until de-asserting TRIG.

It is noteworthy that for SAMP, it is imperative for it to be a *clk* pulse at the *pclk* rate with an average delay relative to *pclk*. This flexibility allows for various methods of generating the SAMP signal. For instance, it could be generated with the *pclk* rising edge, just before its falling edge, or in the middle. Different SAMP waveforms result in different DELAY values. Figure 45 shows multiple valid implementations of the SAMP signal. Note that all the presented signals arrive at *pclk* rate with different phases, and from time to time, introduce an offset.

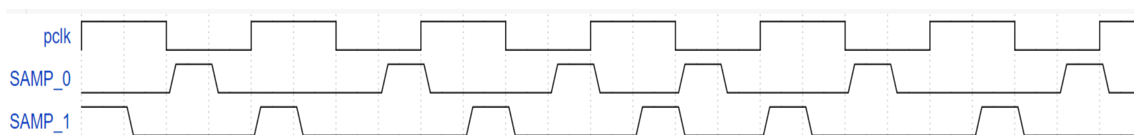

Figure 45. Examples of implementation of the SAMP signal.

The phase of the generated SAMP signal will be randomized, at the start of the simulation and for every RESET_N or PRESET_N signal, between 0 and the pulse width of *pclk*. That is; SAMP must be asserted at any point after the *pclk* rising edge, right away or at the end. To maintain an offset average, each time a SAMP pulse is sent, a delay of 1 *clk* cycle will be added with a 50% probability, making the offset constant on average. A custom driver component on the **SAMP** agent will manage the phase and offset every time it receives a sequence item where SAMP is enabled.

PCLK contains five sequences: two for enabling and disabling the clock respectively, another one that randomizes the value of the measurement duration, then one that asserts TRIG, and one sequence to perform a reset. This final reset sequence will trigger a new randomization of the SAMP phase. These sequences are simple, and they are meant to be used on a higher sequence altogether. By reducing each action to its simplest sequence form, more control over how signals are driven is provided.

Since the **CLK** agent mostly monitors signals, it contains just two basic sequences: one to enable RESET_N and another one to disable it.

The **SAMP** agent only contains a single sequence; enabling the SAMP signal. The handling of the phase and the offset is left to its modified driver component.

The sequences of the **PCLK** agent are all grouped into two top sequences. The first one is called **DoCalibration**. In this case, the measurement duration is randomized, and the block is enabled; afterwards, the calibration is started with the TRIG. Finally, the sequence waits for the expected measurement duration and the block is disabled subsequently. On the other hand, a second sequence named **DoStart** performs the relay operation. The block is again enabled at first, and the START signal is generated. Finally, the block is disabled.

Finally, a top sequence called **DriveSamp** is meant to generate the SAMP signal for a given number of *pclk* cycles. This sequence is meant to be executed in the background until the end of the simulation.

This testbench will run a single test (**TstRandom**) that randomizes the activity of the block as shown in Figure 46.

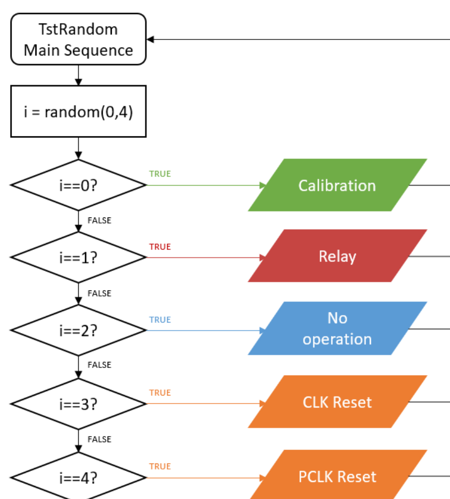

Figure 46. State diagram for the random test of the Chirp Start testbench.

All the actions are modelled as sequences, despite “no operation” not requiring any kind of stimuli. In this context, the test will perform N actions, choosing randomly in each iteration which action to perform.

Since “no operation” does not bring that much significant value, its probability will be set at 3%. It is still required to analyse how the system behaves in its IDLE state. Then, although the relay operation is the main goal of the block, the calibration presents a higher level of complexity, and it is the basis for the correct functioning of the block. For this reason, a calibration operation will take place with a probability of 60%, while the relay action will use 35%.

The remaining 2% will be dedicated to PRESET_N and RESET_N equally. It could be considered an elevated percentage for reset sequences but given that when they are triggered the SAMP signal is randomized, they present a significant opportunity to cover multiple cases in just one simulation run.

6.4 Test results

This section has notable importance not only to this document and the development of the project but also because this design belongs to a marketable integrated circuit. The identification and classification of errors are crucial for the final product to work adequately.

In this case, there are no safety nets as to which errors do exist, and thus exhaustive examination for failure scenarios is required. The following results are based on the PyUVM testbench.

ERROR #1: ACTIVE signal during calibration is asserted for more cycles than expected.

The first error found is shown in Figure 47. It can be observed that the block is enabled correctly, and the DURATION signal is set to 3. At 29.5 ns the TRIG signal starts the calibration, and multiple commutations of SAMP take place, matching the waveform described in the specifications.

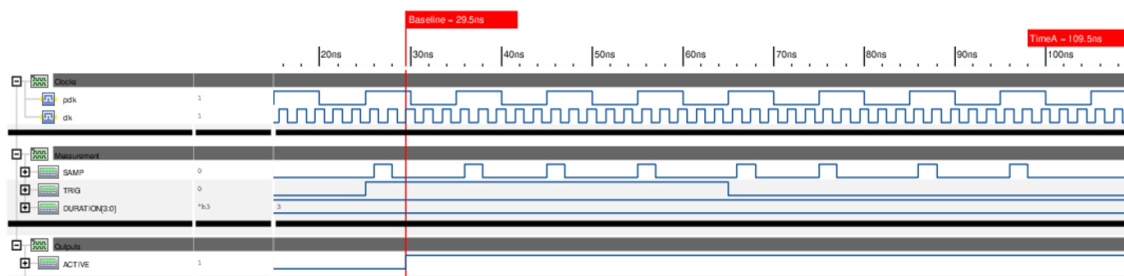

Figure 47. Calibration measurement. Assertion error on ACTIVE.

An assertion error is triggered around 109.5 ns, related to the ACTIVE signal; it is still asserted. To discard an error in the model, the waveform in Figure 47 was checked and the duration of the calibration predicted was correct. The error is located within the design.

Because the number of values allowed for the DURATION signal is not elevated, all cases have been evaluated, and all of them trigger the previous assertion error. Nonetheless, if the simulation is allowed to continue as in Figure 48, disabling the previous assertion, it is possible to observe how the ACTIVE signal returns to 0 after a couple cycles more. Thus, the error is not a matter of ACTIVE never being de-asserted again, but that it expands more cycles than expected.

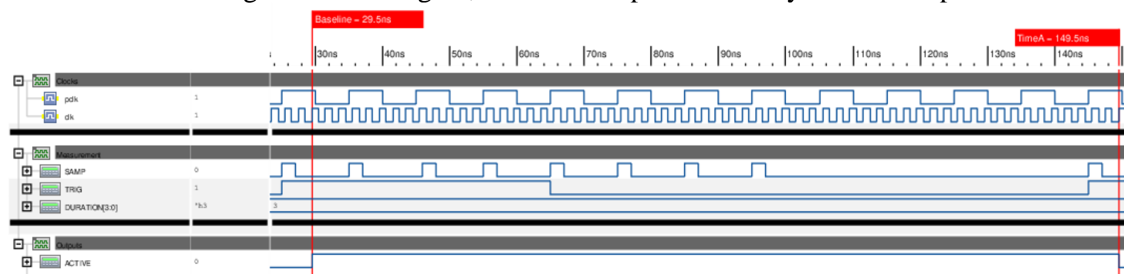

Figure 48. Calibration measurement. Expanded simulation.

The error was correctly reported to the designer for future fixes, but further simulation runs proved another existing scenario in the verification process.

Notice that, so far, the sequence being run is **DoCalibration**, which disabled the block after the measurement. This scenario affects the ACTIVE signal as if it could not change its value whenever the block is not enabled. This explains why, in Figure 48, ACTIVE is set back to 0 after the block is enabled again. Additionally, Figure 49 shows the correct behaviour.

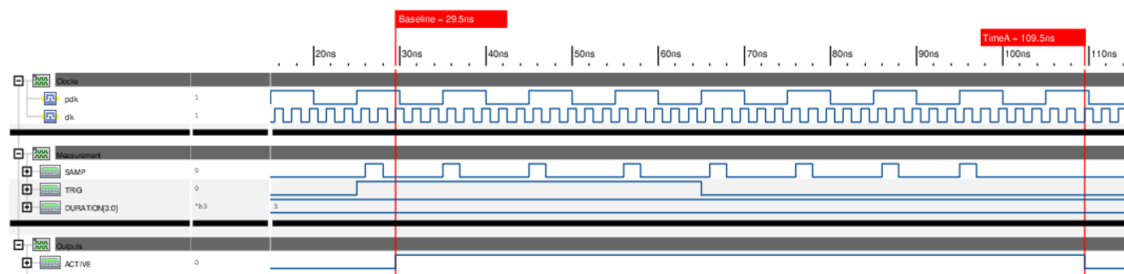

Figure 49. Calibration measurement. Correct behaviour of ACTIVE.

In this case, the problem did not lie within the design itself, but on the specifications of the block. The ENABLE signal is meant to be activated rarely, if not just once every time the IC starts. While this scenario was not thought of as a use case, the specifications did not forbid it, thus the error was triggered. The reader should be reminded that a design must be verified for all possible, not forbidden, cases.

ERROR #2: ACCUM signal value is not frozen until the next measurement.

According to specifications, once the measurement has finished, a new DELAY signal is calculated using an equation that factors the ACCUM signal. However, the accumulator value of each measurement remains stable until a new measurement starts. Note that this situation requires TRIG to be asserted again.

Figure 50 shows a measurement of the same kind as Figure 49, but the ACCUM signal has been added. For each *clk* cycle, the accumulator value is compared against the reference model. The calculations during the measurement are correct. Nonetheless an assertion error rises at 109.5 ns.

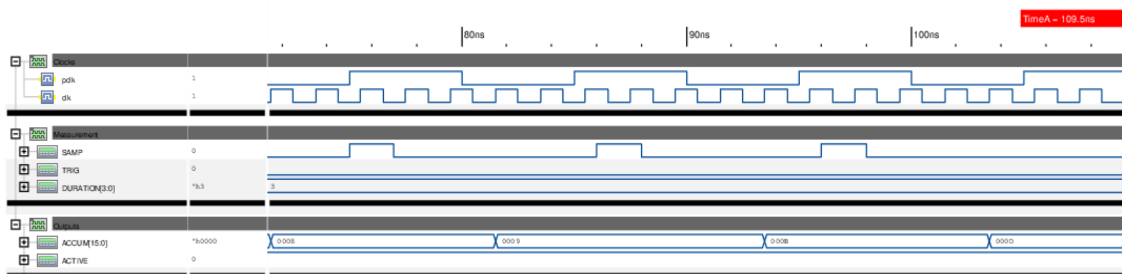

Figure 50. Calibration measurement. Accumulator values.

The value of ACCUM is set back to 0 following the ACTIVE signal being de-asserted. This observation does not match the expected behaviour, as the accumulator value should remain stable (or "frozen") until the start of the next measurement. It is possible to conclude that this waveform represents a design error.

This error was noted and fixed by the designer on the following iteration of the design.

ERROR #3: RAMP signal not asserting correctly.

The following error shows the intricacies of clock domain crossing designs, and how specific situations can trigger errors in a design that works correctly for most scenarios.

At the waveform observed in Figure 51, a reset on the *pclk* domain takes place. It does not affect the DELAY value as this is managed by *clk*, thus the previous value is kept (0x1). Additionally, the SAMP signal is constant at *pclk* rate, and it is resampled correctly into rs_SAMP. Finally, the START signal is asserted at 171,995 ns and resampled into rs_START.

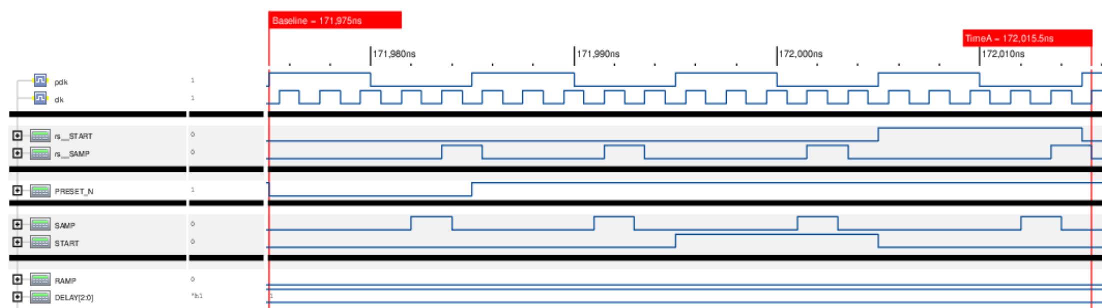

Figure 51. Chirp Start error regarding missing RAMP signal after a reset.

Note that the resampled signals are not used for comparison but are shown in the waveform to let the reader understand the scenario.

According to specifications, the value of rs_START is registered during a rs_SAMP high level. Then, the latest value registered is replicated on the RAMP output signal at the next *clk* rising edge. Regardless, the design description asserts that, if all signal requirements are met (pulse rate and width), setting START to a high level will result in the RAMP signal also being set to 1.

The assertion error arises at 172,015 ns. During the time window in which rs_SAMP is at its high level, the value of rs_START changes from 1 to 0, thus only recording this last value. At the same point, a *clk* rising edge arrives and the model predicts that RAMP should be asserted, but it is not and will not be. The rs_START signal is already set back to 0 and the opportunity to capture its high-level value has already been missed.

Technically, the sampling mechanism is working as intended but the START signal is lost, something that the design should not allow. Both SAMP and START match specifications regarding pulse rate and width, and therefore the issue lies in the sampling mechanism itself. While it works for almost any scenario, this situation triggers unexpected behaviour. There is an intricate issue in how the design operates.

The observed scenario can also be replicated not only after a reset but also after calibration, as shown in Figure 52, indicating that it is an issue that can realistically happen over the course of

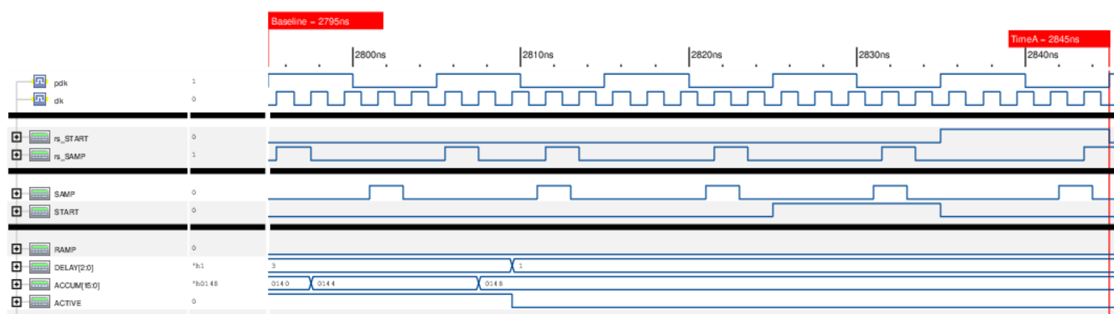

Figure 52. Chirp Start error regarding missing RAMP signal after a calibration.

the system usage.

This error serves as an example of how the verification process not only finds errors related to the implementation of a design but to its concept. The former is an issue that can be primarily solved by re-writing the HDL code, the latter requires restructuring the internal mechanism.

6.4.1 Discussion

This DUT is significantly more complicated when it comes to its functioning compared to the previous designs. Nonetheless, the testbench was executed correctly and multiple errors were found, including a category not previously explored: behaviour missing in the specifications.

This chapter, while may present a less convoluted testbench than the one observed for the bus arbiter, proves that PyUVM and cocotb are suitable for the verification of real designs being developed in the industry. The testbench was able to successfully detect three scenarios where the design was failing, thus avoiding the propagation of these errors to future stages in the development process.

Although there is no SV/UVM implementation to compare, the previous chapters provide information on this topic. The purpose of this chapter was to solely focus on the verification of a real design.

Chapter 7. Summary

The present chapter aims to provide a clear and comprehensive conclusion to the present document. Using the objective data obtained in each of the three verification scenarios, the obtained results for *PyUVM* and *SV/UVM* will be compared. Then, analysing the user experience as well as each used programming language, further distinctions between both approaches will be drawn. Subsequently, notes on the expected usage of *cocotb* in the verification industry will be described.

Finally, the initial goals of the thesis will be reviewed and thoughts about this new digital verification technique will be written. Additionally, ideas on how *cocotb* can be further developed and used in the verification industry will be discussed.

7.1 Comparison

This subsection presents an objective comparison between PyUVM and SV/UVM based on the obtained results for the SFIFO and the bus arbiter testbenches. This comparison should reflect an unbiased point of view. Additionally, a personal evaluation can also be found.

As observed in previous chapters, the number of **code lines** needed to implement a testbench in SV/UVM compared to PyUVM is significant. For instance, the SFIFO testbench required approximately 30% more lines of code. Although it has been discussed the fact that there exist multiple ways of writing the same statement, there is a point to be considered. The UVM framework is considered a high-verbose approach to verification, and certain structures must be defined every time, generating what is known as "boilerplate" code; code that does not help to implement any functionality but instead is required for the testbench structure to work. The syntax complexity of SystemVerilog amplifies this, resulting in more line codes, and thus generating more points of failure for future maintainability.

Regarding **runtime**, each DUT has been simulated on the same dedicated machine, sharing an equal UVM testbench structure, with the same stimuli generator and for multiple executions of varied iterations. Additionally, the random seed has been fixed for all simulation runs. These conditions are considered enough for this metric to be regarded as objective.

In every verification scenario, for all number of iterations, the observed runtime in SV/UVM-based testbenches is lower than the one presented by PyUVM. Figure 53 shows the average runtime difference between both approaches for each verified DUT, normalized to the slowest implementation. Note that this average includes the three possible numbers of iterations. The enhanced implementation of the PyUVM testbench is not shown for the SFIFO.

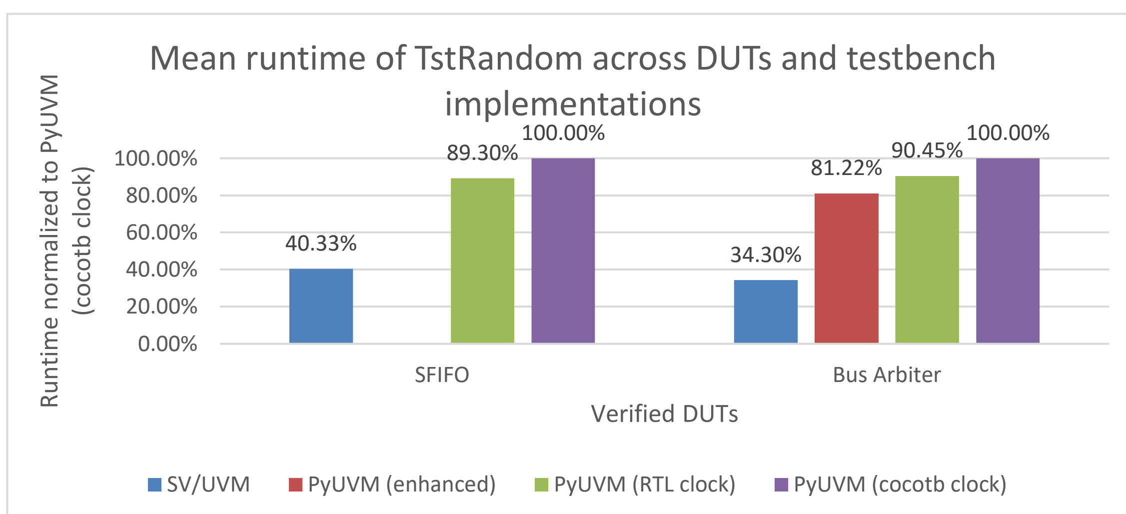

Figure 53. Mean runtime of TstRandom across DUTs. Normalized to the respective slowest implementation.

This result does not come as a surprise. As explained in Chapter 3, the fact that simulators can directly interpret code written in SystemVerilog allows for better performance, since the cocotb framework must use a dedicated interface between the simulator and Python. Even when PyUVM has been optimized as shown in the enhanced approach, the observed runtime is 47% slower than its SV/UVM counterpart.

From this point onwards, the following metrics are based on the experience acquired while working with PyUVM and cocotb. Although the subsequent statements are validated with references, these metrics hold a different type of value from the previous ones, based more on the user experience rather than the frameworks themselves.

An example of this is the **learning curve**. Both verification and coding skills are independent of this comparison since both are required to use either PyUVM or SV/UVM. For this reason, the analysis revolves around the used programming languages. In this context, the number of users for each one could merely represent popularity, and the extension of its documentation could reflect the available capabilities. Additionally, SystemVerilog is not a programming language as such, but an HDVL, making it even harder to compare.

Python is a general-purpose language taught across diverse professional and academic fields. Its low level of syntax complexity provides a comprehensive environment where scripts can be designed and run easily. On the other hand, SystemVerilog is a language exclusively dedicated to the design and verification of circuits. This feature of SystemVerilog allows its users to use specific structures and functions related to the application field. Nevertheless, to extract the maximum performance out of these structures and functions, the user must have a significant level of knowledge of the language and must cope with the unavailability of data access techniques compared to Python.

While SystemVerilog presents powerful tools that specifically target verification, these tools require more time and dedication than using Python-based testbenches.

When it comes to **integration**, Python is one of the best-known scripting languages. The considerable number of available third-party libraries, and its popularity as a programming language, allow for multiple ways to manage the simulation data and perform different actions over it. Nonetheless, most EDA tools are based on the premise of using SystemVerilog or other HDLs. For this reason, this comparison has two different approaches.

For custom tools or scripts to analyse the performed simulation, Python represents the best option. An example of this could be a script that performs statistical analysis of the obtained data or building a dedicated regression manager.

In the case of connecting with other EDA tools of the same vendor, and in specific cases with other vendors, SystemVerilog may be the best approach. It is one of the standard languages of the verification industry and thus, usable across different programs.

The **maintainability** of each approach lies again in their programming language. The lower syntax complexity of Python allows for easier-to-read code, thus increasing maintainability. As discussed in the line count comparison, the more statements required to define a functionality, the more failure points are introduced, and the more readability is compromised. Python is a language whose syntax rules allow the user to quickly understand how the code operates. A simple but effective example of this is the implementation of the logical AND operator as “and” instead of “&&”. While most programmers are used to the latter notation, the former makes Python a readable programming language that quickly describes the implemented functionality.

Finally, in the **industrial adoption** category, the most relevant is easily SystemVerilog. As commented previously, it is the default language of significant EDA vendors (besides VHDL). This situation results in the verification industry and verification environments being dominated by SystemVerilog.

Nonetheless, cocotb has been used by the CERN on the ATLAS experiment [23] for three different ASICs. Although this last example used the plain cocotb library, the sponsorship of PyUVM by a notable EDA vendor such as Siemens is an indication of the growing interest in the industrial application of this technology.

7.1.1 Limitations and target cases

The most significant problem for cocotb, and its derived libraries, is the runtime. This issue has been observed in sections 4.4.1 and 5.4.1 At the same time, its most powerful characteristic is its simplicity thanks to the Python programming language. This scenario presents a clear trade-off between time spent in the simulation itself and designing and maintaining the testbench.

For a defined runtime, if there is an error to be fixed, the overall time spent in verification is exponentially higher. The error must be found, documented, fixed, and simulated again. The more time is spent in simulation, the more the fixing process is delayed. The repercussion of a higher runtime has significant ramifications.

In this sense, PyUVM could present a better option for complex designs that require intricate reference models. Its simplicity allows, as seen in the arbiter chapter, to recreate the target's design behaviour rapidly and with simple statements. However, testbenches with significant coverage requirements need more tests to be run and thus more simulation time, not being the best scenario for PyUVM.

It can be said then, that while PyUVM and cocotb present a higher runtime when it comes to the simulation itself, the simplicity of writing and running the testbenches makes these frameworks capable of managing the verification of complex designs. Additionally, the maintainability of Python-based testbenches, as discussed before, can also save time when readjusting testbench parameters for an updated version of the design, adding functionalities, or modifying the environment.

7.1.2 Performance enhancing

The GitHub repository of cocotb presents a discussion regarding methods to improve the performance of the testbenches. The most significant proposals are the use of fewer triggers, as well as not changing values constantly.

In its functioning, every trigger or change of value presents a call to the simulator interface. This call is costly because the simulator stops, communicates through the interface, and does not restart until the data has arrived at the Python code. The less transactions, the better the runtime.

This scenario can be observed with the enhanced version of PyUVM for the arbiter. Because the change of multiple signals is done at the same time, the observed runtime is less than the original version.

7.2 Goal review

In this subsection, the initial goals described in section 1.2 will be reviewed, and their success will be assessed based on the different chapters of this document.

In the first place, an examination of the current challenges of SV/UVM-based testbenches has been described in Chapter 2. A look was taken at how complex reference models slow the verification process, and how new engineers may find an additional barrier in the SystemVerilog language. Additionally, a case for Python-based testbenches was made with reasonable statements.

Another goal was to provide insight into the functioning of cocotb, how it is possible for Python code to interact with simulators, and the basic details on how to write and build testbenches using the plain cocotb library and the PyUVM framework. This information has been extensively covered in Chapter 3.

The most significant goal was the use of this new Python-based framework for the verification of different DUTs that represent three different scenarios (a ramp-up project, a more complex testbench, and an industrial application). This goal has been completed successfully in Chapter 4, Chapter 5, and Chapter 6.

The main goal is covered in this chapter as well as the previous ones. The purpose of this thesis was to evaluate the viability of PyUVM testbenches in the verification industry. This has been accomplished through its comparison to the already stablished SV/UVM approach, and its application to a design currently being developed by a notable company (Bosch) in the semiconductor industry In this sense, this goal can be considered covered, and thus, the thesis purpose has been completed.

7.3 Concluding thoughts

The usage of this framework during the length of six months has provided invaluable knowledge about its usage and viability. Referring to its limitations and target cases, *cocotb* can be considered a framework that works as expected, that accomplishes its purpose of using Python as a verification language, and that its main disadvantage is a matter of optimization.

This last point, optimization, may not lie entirely on the cocotb side but on the used interface. Since the simulators are predominately used with SystemVerilog, it is possible that the usage of interfaces like VPI has not been optimized, although this is an assumption.

Nonetheless, although it is possible that the runtime cannot be matched any time soon, *cocotb* presents a notable alternative to the verification industry. New engineers, or other professionals not related to the verification of digital designs, can find an easier way to get started in the design of testbenches. This allows for a bigger market of candidates as well as reducing the training time of new engineers; they are only confronted with learning the concepts of verification instead of also learning a new language such as SystemVerilog.

Overall, with its defined target cases and limitations, cocotb and PyUVM can be considered a notable tool to be used for digital verification.

7.4 Future work

The next significant scenario in the IC industry revolves around unifying verification with validation.

As of now, the pre-silicon verification of a design (simulation) is performed by a specific team with its own testbench, whereas the post-silicon validation (tests on a physical prototype) is conducted by another department with its specialized tests. These tests differ as the physical prototype requires inputs from a signal generator, and its outputs can be monitored in an oscilloscope, for example.

Available languages like SystemC can bridge this problem, as they allow to drive C-based interfaces of the physical testers. Nonetheless, it could be feasible to implement a similar structure with Python, as there are already libraries that allow to control these instruments; a framework from the ground up to relate cocotb testbenches to testers. The advantage over SystemC has the same basis as why PyUVM is a valid alternative to SV/UVM. Syntax simplicity and popularity of the language allow for newcomers to focus on the verification/validation process instead of dedicating useful time to overcome the barrier that languages like C, C++, or SystemVerilog present.

Although it is not the same to control a simulator as to manage instruments like an oscilloscope, an intermediate framework based on cocotb could make it possible. In this sense, a single testbench could be used for both pre- and post-silicon verification. This scenario allows the design to be held under the same expectations before and after its production, presenting the culmination of reproductivity and reusability.

Further future work may involve optimizing the cocotb framework to reduce runtime, as well as adding new features or creating new related projects that may help other engineers, like a regression manager.

For applications outside the scope of improving the technology, cocotb and PyUVM could become a notable approach for the verification of digital designs in the semiconductor industry.

7.5 SDG alignment

This project aligns with several points described in the *United Nations* (UN) *Sustainable Development Goals* (SDG) [34].

One of the topics of this thesis is to provide the semiconductor industry with an understanding of a new technology, thus fostering innovation in the field. For this reason, the project aligns with goal number nine (industry, innovation and infrastructure).

At the same time, a successful verification process avoids the fabrication of faulty products, increasing the efficiency of the manufacturing process. Thus, presenting cocotb as an improvement or expansion of existing verification environments aligns with goal number twelve (responsible consumption and production).

Finally, this technology presents a smoother learning curve compared to SV/UVM while at the same time being able to obtain the same test results. This provides a friendly environment for new engineers as well as maintaining the validity of the tests. In this context, Chapter 3 and the subsequent verification of multiple DUTs align with goal number eight (decent work and economic growth).

References

1. **Foster, Harry.** Part 8: The 2020 Wilson Research Group Functional Verification Study - Verification Horizons. *Siemens Digital Industries Software.* [Online] Siemens Software, 6 January 2021. [Cited: 26 February 2024.] https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/.

2. **IEEE.** *IEEE Standard for Verilog Hardware Description Language.* s.l. : IEEE, 2009.

3. **IEEE.** *IEEE Standard for VHDL Language Reference Manual.* s.l. : IEEE, 2008.

4. **IEEE.** *IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language.* s.l. : IEEE, 207.

5. **Foster, Harry.** Part 10: The 2022 Wilson Research Group Functional Verification Study - Verification Horizons. *Siemens Digital Industries Software.* [Online] Siemens Software, December 26, 2022. [Cited: February 26, 2024.] https://blogs.sw.siemens.com/verificationhorizons/2022/12/26/strongpart-10-the-2022-wilson-research-group-functional-verification-study-strong/.

6. **Python Software Foundation.** Documentation. *Python.* [Online] [Cited: February 28, 2024.] https://www.python.org/doc/.

7. **Python Software Foundation.** *PyPi - The Python Package Index.* [Online] [Cited: February 26, 2024.] https://pypi.org/.

8. **FOSSi Foundation.** *cocotb | Python verification framework.* [Online] FOSSi Foundation. [Cited: February 26, 2024.] https://www.cocotb.org/.

9. **cocotb community.** Contributors. *cocotb 1.8.1 documentation.* [Online] [Cited: February 26, 2024.] https://docs.cocotb.org/en/stable/contributors.html.

10. **Wagner, Philipp.** Announcing the cocotb unconference at ORConf. *cocotb.* [Online] August 2, 2023. [Cited: February 26, 2024.] https://www.cocotb.org/2023/08/02/orconf.

11. **Salemi, Ray.** pyuvm/pyuvm. *GitHub.* [Online] [Cited: February 28, 2024.] https://github.com/pyuvm/pyuvm.

12. **IEEE.** *IEEE Standard for Universal Verification Methodology Language Reference Manual.* s.l. : IEEE, 2020.

13. **Taraate, Vaibbhav.** *ASIC design and synthesis: RTL design using Verilog.* s.l. : Springer, 2021.

14. *Design-Flow and Synthesis for ASICs: a case study.* **Bombana, Massimo.** s.l. : Proceedings of the 32nd annual ACM/IEEE Design Automation Conference, 1995.

15. *ASIC Design Flow And Methodology - An Overview.* **Shetty, Ashish A.** 7, s.l. : SSRG International Journal of Electrical and Electronics Engineering, 2019, Vol. 6.

16. **Mehta, Ashok.** *ASIC/SoC functional design verification.* s.l. : Springer, 2018.

17. **Alsop, Thomas.** Semiconductor companies market revenue share worldwide 2008-2023. *Statista.* [Online] January 2019, 2023. [Cited: February 28, 2024.] https://www.statista.com/statistics/266143/global-market-share-of-leading-semiconductor-vendors/.

18. **Foster, Harry.** Part 9: The 2022 Wilson Research Group Functional Verification Study - Verification Horizons. *Siemens Digital Industries Software.* [Online] Siemens Software, December 18, 2022. [Cited: February 28, 2024.] https://blogs.sw.siemens.com/verificationhorizons/2022/12/18/part-9-the-2020-wilson-research-group-functional-verification-study-2/.

19. **Cerny, Eduard.** *SVA: The Power of Assertions in SystemVerilog.* s.l. : Springer, 2015.

20. **Height, Hannibal.** *A practical guide to adopting the universal verification methodology (UVM).* s.l. : Cadence Design Systems, 2013.

21. **FOSSi Foundation.** cocotb/cocotb. *GitHub.* [Online] [Cited: February 28, 2024.] https://github.com/cocotb/cocotb.

22. **cocotb community.** Simulator Support. *cocotb 1.8.1 documentation.* [Online] [Cited: February 28, 2024.] https://docs.cocotb.org/en/stable/simulator_support.html.

23. **Rosser, Ben.** Cocotb: a Python-based digital logic verification framework. *CERN.* [Online] December 11, 2018. [Cited: February 28, 2024.] https://indico.cern.ch/event/776422/attachments/1769690/2874927/cocotb_talk.pdf.

24. **cocotb community.** cocotb Internals. *GitHub.* [Online] [Cited: February 28, 2024.] https://github.com/cocotb/cocotb/wiki/cocotb-Internals.

25. **Python Software Foundation.** Python/C API Reference Manual. *Python Documentation.* [Online] [Cited: February 28, 2024.] https://docs.python.org/3/c-api/index.html.

26. **cocotb community.** How does cocotb work? *cocotb 2.0.0 documentation.* [Online] [Cited: February 28, 2024.] https://docs.cocotb.org/en/latest/#how-does-cocotb-work.

27. **cocotb community.** Writing Testbenches. *cocotb 1.8.1 documentation.* [Online] [Cited: February 28, 2024.] https://docs.cocotb.org/en/stable/writing_testbenches.html.

28. **cocotb community.** Reference Card. *cocotb 1.8.1 documentation.* [Online] [Cited: February 28, 2024.] https://docs.cocotb.org/en/stable/refcard.html.

29. **cocotb community.** Build options and Environment Variables. *cocotb 1.8.1 documentation.* [Online] [Cited: February 28, 2024.] https://docs.cocotb.org/en/stable/building.html#makefile-based-test-scripts.

30. **tpoikela.** tpoikela/uvm_python. *GitHub.* [Online] [Cited: February 28, 2024.] https://github.com/tpoikela/uvm-python.

31. **Salemi, Ray.** *Python for RTL Verification: A Complete Course in Python, Cocotb, and PyUVM.* s.l. : Independent., 2022.

32. **Salemi, Ray.** PYUVM - Verification Horizons. *Siemens Digital Industries Software.* [Online] Siemens Software. [Cited: February 28, 2024.] https://blogs.sw.siemens.com/verificationhorizons/category/tips-tricks/pyuvm/.

33. **ARM Ltd.** *AMBA APB Protocol Specification.* s.l. : ARM, 2023.

34. **United Nations.** Sustainable Development. *United Nations.* [Online] [Cited: February 28, 2024.] https://sdgs.un.org/goals.