FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Development of a reconfigurable multi-protocol verification environment using UVM methodology

Pedro Araujo

WORKING VERSION

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Teacher supervisor: José Carlos Alves

Company supervisors: Luis Cruz and Domingos Terra

June 30, 2014

© Pedro Araujo, 2014

Resumo

Com o contínuo avanço da tecnologia CMOS, bem como, com o crescimento dos custos de produção, torna-se cada vez mais importante produzir circuitos que cumpram todas as especificações á primeira (*first time right*). Uma má verificação pode conduzir a que a janela de mercado se feche, enquanto se corrigem os erros detectados da primeira implementação.

O tipo de testes que têm de ser feitas para os diferentes protocolos é em grande parte similar, apesar das especificidades inerentes a cada um. A metodologia de verificação UVM permite estruturar o ambiente de verificação numa arquitectura configurável baseada em camadas, utilizando diferentes blocos genéricos. Desta forma é possível especializar o ambiente de verificação a um dado protocolo adicionando apenas as funcionalidades necessárias à camada mais próxima do DUT (*Device Under Test*).

O UVM é uma metodologia que segue um standard que foi criado pela Accellera em conjunto com os maiores fabricantes de ferramentas de desenho de circuitos electrónicos (Synopsys, Mentor, Cadence). Esta parceria pretende estabelecer uma uniformização na indústria electrónica, aumentando a eficiência do desenvolvimento e reutilização dos ambientes de verificação usados. O UVM é suportado pelas principais ferramentas de verificação/simulação de circuitos digitais, o que o torna compatível com qualquer um dos simuladores das referidas empresas. A Accellera disponibiliza para download uma API (Application Programming Interface) e uma implementação de referência (baseado numa biblioteca de classes em SystemVerilog (IEEE 1800)), que suportam o desenvolvimento de um ambiente de verificação genérico.

A dissertação origina da necessidade de reduzir o tempo de construção de um ambiente de verificação para um novo projecto. Tendo esta necessidade em consideração, a presente dissertação foca-se na tarefa principal de desenvolver um ambiente de verificação genérico que suporte arquitectura básica de vários protocolos de comunicação, e a reconfigurabilidade necessária para suportar múltiplas configurações do mesmo DUT.

Este projecto foi sugerido pela Synopsys, uma das empresas líder na indústria da Electronic Design Automation, e irá ser executado no contexto do trabalho realizado pela Synopsys com protocolos de comunicação a alta velocidade. O project vai ser realizado com as ferramentas da Synopsys para a simulação e execução de código Verilog e SystemVerilog.

O resultado do trabalho feito para esta dissertação irá consistir num ambiente de verificação genérico desenvolvido em SystemVerilog e seguindo a metodologia UVM. Juntamente com o ambiente desenvolvido, esta dissertação serve de documentação sobre as funções e classes criadas e a relação establecidas entre elas.

Abstract

With the ongoing progress of the CMOS technology, as well with the growing of production costs, it's more important than ever to develop digital circuits that comply with the specifications at the first try (*first time right*). An incomplete verification can lead to a closing market window while the errors detected during the first implementation are still being fixed.

The sort of tests that have to be done for the different communication protocols are very similar between them, in despite of the nature of each one. The UVM verification methodology allows to structure a verification environment in a configurable architecture based on layers by using a variety of generic blocks. Thereby, it's possible to specialize a verification environment to a given protocol just by adding the necessary functionalities to the layers closer of the DUT (*Device Under Test*).

UVM is a methodology that follows a standard created by Accellera jointly with the biggest companies in the industry of electronic design automation (Synopsys, Mentor, Cadence). This partnership intends to establish a standardization for verification methodologies in the electronic industry, increasing the effiency of the development and the reusability of the employed verification environments. UVM is used by the main verification/simulation tools of digital circuits and therefore, it's compatible with any of the simulators of the mentioned companies. Accellera provides an API (Application Programming Interface) and an implementation (based on a library of classes in SystemVerilog (IEEE 1800)) which supports the development of verification environments.

This dissertation stems from the need in reducing the set up time of the verification environment for a new project. By having this need in consideration, the present dissertation focus its main goal in the development of a generic verification environment that supports the core architecture of multiple communication protocols, and the necessary reconfigurability to support multiple configurations of the same DUT.

The project was suggested by Synopsys, one of the leading companies in the Electronic Design Automation industry, and it will performed within the scope of Synopsys' work with high-speed communication protocols. This project will be assisted by Synopsys' tools for the simulation and the execution of Verilog and SystemVerilog code.

The result of work done for this dissertation will consist on a generic verification environment written in SystemVerilog while following the UVM methodology. Along with the developed environment, this dissertation provides a thorough documentation about the functions and classes created and the relationship between them.

Acknowledgments

This project took more than 5 months to complete, 5 months of long working hours and of some nights of short sleep. But it helped to have good people supporting me and keeping me on track, specially during the hardest times.

Inside of this circle of people, I would like first to thank the people who proposed this project in the first place: Luis Cruz and Domingos Terra. They had an incredible patience with me by answering to all my questions whenever I needed, and the help that they provided me was essential. I would also like to thank the professor José Carlos Alves, who was also part of the team and who supervisioned me during the development of the project.

I would like to give my thanks to some friends who accompanied me during the long hours spent at the office. They provided me with some good talks, which helped me to relax and to get a new perspective whenever I got stuck in some part of the project: Denis Silva, Helder Campos and Henrique Martins.

I want to include here, as well, the guys from the analog team: Bruno Silva, Hugo Gonçalves and Patricio Ferreira. Their presence during the lunch and snack breaks was the most enjoyable and their cookies, the tastiest.

And at last, but not the least, I would like to thank some special friends, friends that showed me and taught me that distance doesn't break friendships. I would like thank Charlotte, Daniel and Fotini. They provided me with some company through some long nights, they reminded me to keep my progress reports updated and they lifted my spirits during the bad days. They were able to give the best of advices when I needed most and their support was the most important to me. Thank you.

I also want to thank all my friends who shared me with some great moments throughout this academic journey.

Pedro Araujo

"All courses of action are risky,
so prudence is not in avoiding danger, but calculating risk and acting decisively.
Make mistakes of ambition and not mistakes of sloth.
Develop the strength to do bold things, not the strength to suffer."

Niccolo Machiavelli

Contents

1 Introduction **1**
1.1 Context . . . 2
1.2 Structure of the document . . . 2

2 State of the art **5**
2.1 Hardware Description Languages . . . 5
2.2 Functional Verification . . . 6
2.3 Hardware Verification Languages . . . 8
2.4 Verification Methodologies . . . 9
2.5 The Universal Verification Methodology . . . 10
2.5.1 UVM Overview . . . 11
2.5.2 UVM Classes . . . 12
2.5.3 UVM Phases . . . 14
2.5.4 UVM Macros . . . 15
2.5.5 Typical UVM class . . . 16
2.5.6 TLM-1: Ports . . . 17
2.5.7 TLM-2.0: Sockets . . . 20
2.6 Conclusion . . . 21

3 Analysis of communication protocols **23**
3.1 X-PHY Overview . . . 23
3.2 X-PHY Verification . . . 27
3.3 I2C Overview . . . 31
3.4 I2C Verification . . . 33
3.4.1 Verifying an I2C slave . . . 34
3.4.2 Verifying an I2C master . . . 37
3.4.3 UVM verification components created for the I2C interfaces . . . 38
3.5 SOC Overview . . . 39
3.6 SOC Verification Plan . . . 41
3.6.1 Testing the slave interface . . . 42
3.6.2 Testing the slave interface and the low-speed lane . . . 42
3.6.3 Testing the slave interface and the low-speed and high-speed lanes . . . 44
3.7 Conclusion . . . 46

4 The Verification Environment **47**
4.1 Testbench Overview . . . 47
4.1.1 Class table . . . 52
4.1.2 File system . . . 54

4.2 Configuration Blocks 55
4.2.1 Agent configuration block 55
4.2.2 Env configuration block 57
4.3 The Test Block 58
4.4 The *Env* Block 59
4.5 The Agent Manager 61
4.6 Generic Info Block 62
4.7 Socket containers 63
4.8 Agents 66
4.8.1 Master Agent 67
4.8.2 Slave Agent 68
4.9 Broadcaster 69
4.10 Monitors 71
4.10.1 Master Monitors 72
4.10.2 Slave Monitors 73
4.10.3 Normal Monitors 74
4.11 Drivers 74
4.12 Scoreboard, Sequencers, sequences and transactions 75
4.13 Work flow 75
4.14 Conclusion 76

5 Application of the Environment to the SOC **77**
5.1 Verification of the SOC 77
5.1.1 I2C Master Agent 78
5.1.2 I2C Slave Agent 81
5.1.3 Grouping the agents 82
5.2 Conclusion 84

6 Application of the Environment to the AC97 **85**
6.1 Overview of the AC97 85
6.1.1 AC-Link Interface 86
6.2 Verification components 88
6.2.1 Driving the inputs of the AC97 audio codec 88
6.2.2 Collecting data items from the AC97's inputs and outputs 90
6.2.3 Evaluating the results of the test 92
6.2.4 Agent manager 92
6.2.5 Summary of the verification components 93
6.3 Test cases 94
6.3.1 First test: Testing the DUT's registers 94
6.3.2 Second test: Testing the Digital to Analog functionality 95
6.3.3 Third test: Testing the Analog to Analog functionality 97
6.3.4 Fourth test: Testing the Analog to Digital functionality 99
6.3.5 Automatization of the environment 100
6.4 Conclusion 100

7 Conclusion **103**
7.1 Summary of the developed work 103
7.2 Features and results of the concluded work 104

A UVM Guide for Beginners **105**
A.1 Introduction . . . 105
A.2 The DUT . . . 107
A.3 Defining the verification environment . . . 108
A.3.1 UVM Classes . . . 110
A.3.2 UVM Phases . . . 111
A.3.3 UVM Macros . . . 112
A.3.4 Typical UVM class . . . 113
A.3.5 SimpleAdder UVM Testbench . . . 114
A.4 Top Block . . . 115
A.5 Transactions, sequences and sequencers . . . 118
A.5.1 Sequence . . . 121
A.5.2 Sequencer . . . 122
A.6 Driver . . . 124
A.6.1 Creating the driver as a normal testbench . . . 126
A.6.2 Implementing the UVM driver . . . 128
A.7 Monitor . . . 131
A.7.1 TLM ports . . . 134
A.8 Agent . . . 137
A.9 Scoreboard . . . 139
A.10 Env . . . 142
A.11 Test . . . 144
A.12 Running the simulation . . . 146

List of Figures

2.1 Generic testbench 6
2.2 Direct testing progress [13, p.6] 7
2.3 Random testing progress [13, p.8] 7
2.4 Structured testbench 8
2.5 Evolution of verification methodologies 10
2.6 Typical UVM testbench 11
2.7 Partial UVM class tree 13
2.8 Partial list of UVM phases 14
2.9 Port-export communication 17
2.10 Analysis port communication 19
2.11 Representation of a socket communication with an initiator and a target component 20
2.12 Socket communication with an initiator, a passthrough and a target component . . 20

3.1 Applications of a possible X-PHY 24
3.2 X-PHY basic lane 24
3.3 States of X-PHY 25
3.4 Representation of 3 different situations for a trade-off between power and performance 25
3.5 X-PHY with 3 lanes 26
3.6 An UVM verification environment for the receiver of X-PHY 27
3.7 An UVM testbench with 2 instances of the same agent for a X-PHY device with 2 Rx 28
3.8 An UVM verification environment with support for agent configuration 29
3.9 An UVM verification environment with support for an agent manager 30
3.10 Typical topology for an I2C interface 31
3.11 Representation of a start and stop condition on an I2C bus 31
3.12 Representation of an I2C read operation 32
3.13 Representation of an I2C write operation 32
3.14 Representation of an I2C write operation of 2 bytes 32
3.15 Representation of an I2C write operation without slave acknowledgement 33
3.16 Representation of a typical I2C timing diagram 33
3.17 Serial and parallel interfaces created for the I2C device 34
3.18 Class tree of the created components for a possible I2C testbench 38
3.19 Overview of the created SOC 39
3.20 Activating the low-speed lane of the SOC 40
3.21 Sending sound samples to the controller 40
3.22 Activating the high-speed lane of the SOC 40
3.23 Sending video samples to the controller 41

3.24 Testbench for one I2C interface 42
3.25 Testbench for two I2C interfaces 43
3.26 Testbench for the SOC with an agent manager 44
3.27 Complete testbench for the SOC 44
3.28 Testbench for the SOC with *Agent Slave 2* disabled 45
3.29 Testbench for the SOC with *Agent Slave 1* disabled 45

4.1 A top level view of the verification environment 48
4.2 A top level view of the verification environment 49
4.3 A top level view of a slave agent 50
4.4 A top level view of a master agent 51
4.5 A top level view of the verification environment with class names 51
4.6 Class tree of the created testbench 53
4.7 Configuration blocks of the verification environment 55
4.8 Test block of the verification environment 58
4.9 *Env* block of the verification environment 59
4.10 Agent manager block of the verification environment 61
4.11 Sockets from the agent manager 63
4.12 Agents of the verification environment 66
4.13 A typical constitution of a master agent 67
4.14 A typical constitution of a slave agent 68
4.15 The broadcaster block 70
4.16 The master monitor block 72
4.17 The slave slave block 73

5.1 Agent for testing an I2C slave interface 78
5.2 Agent for testing an I2C master interface 81
5.3 Overview of the complete verification environment for the SOC 82
5.4 Verification environment reconfigured for a revision of the SOC that features only one I2C-Slave 82
5.5 Verification environment reconfigured for a revision of the SOC that features one I2C-Slave and one I2C-Master 83

6.1 A simple model of LM4550 [11, p. 2] 85
6.2 Codec Input frame of an AC-Link interface 86
6.3 Codec Output frame of an AC-Link interface 87
6.4 LM4550 registers highlighted 94
6.5 LM4550 Testbench for the registers 95
6.6 LM4550 digital to analog functionality highlighted 95
6.7 LM4550 Testbench for the digital to analog functionality 96
6.8 LM4550 Testbench for the digital to analog functionality with the agent manager 97
6.9 LM4550 analog to analog functionality highlighted 97
6.10 LM4550 Testbench for the analog to analog functionality 98
6.11 LM4550 analog to digital functionality highlighted 99
6.12 LM4550 Testbench for the analog to digital functionality 99

A.1 Representation of the DUT's inputs/outputs 107
A.2 Operation of the DUT 107
A.3 Typical UVM testbench 109

A.4 Partial UVM class tree 110
A.5 Partial list of UVM phases 111
A.6 SimpleAdder Final Testbench 114
A.7 Relation between a sequence, a sequencer and a driver 119
A.8 State of the verification environment after the sequencer 122
A.9 Driver waveform 128
A.10 State of the verification environment with the driver 130
A.11 State of the verification environment after the monitors 133
A.12 Port-export communication 134
A.13 Analysis port communication 135
A.14 State of the testbench after the agent 138
A.15 Usage of FIFO in the scoreboard 139
A.16 State of the testbench after the scoreboard 141
A.17 State of the testbench after the env 143
A.18 Final state of the testbench 145

List of Tables

2.1 Sample code for ports and exports 18
2.2 Sum up of TLM-1.0 ports 19

3.1 I2C transaction 35
3.2 I2C Agent Config 36
3.3 I2C verification components 38
3.4 I2C verification components 41

4.1 Elements of the class *generic_agent_config* 52
4.2 Elements of the class *generic_agent_config* 56
4.3 Elements of the class *generic_env_config* 57
4.4 Elements of the class *generic_test* 58
4.5 Elements of the class *generic_env* 60
4.6 Elements of the class *generic_agent_manager* 62
4.7 Elements of the class *generic_info_block* 63
4.8 Elements of the class *socket_slave_container* 64
4.9 Elements of the class *socket_slave_container* 65
4.10 Elements of the class *generic_agent_master* 67
4.11 Elements of the class *generic_agent_slave* 69
4.12 Elements of the class *broadcaster* 70
4.13 Elements of the class *broadcaster* 72
4.14 Elements of the class *broadcaster* 73

5.1 I2C-Master transaction 80
5.2 Verification components for the testbench of the SOC 84

6.1 Transaction for generating values for the sine generator - *ac97_trans_base* 89
6.2 Transaction for generating values for the sine generator - *ac97_trans_sine* 90
6.3 Transaction for collecting transactions from the codec's outputs - *ac97_trans_ana* 92
6.4 Elements of the AC97 testbench 93

A.1 Sample code for ports and exports 135
A.2 Sum up of TLM-1.0 ports 136

Abbreviations and Symbols

API	Application Programming Interface
AVM	Advanced Verification Methodology
DUT	Device Under Test
EDA	Electronic Design Automation
eRM	e Reuse Methodology
HDL	Hardware Description Language
HVL	Hardware Verification Language
IEEE	Institute of Electrical and Electronics Engineers
OVM	Original Verification Methodology
PSL	Property Specification Language
PHY	Physical Layer of the OSI model
SOC	System on a Chip
RTL	Register-transfer level
RVM	Reuse Verification Methodology
TLM	Transaction Level Modeling
URM	Universal Reuse Methodology
UVM	Universal Verification Methodology
VHDL	VHSIC Hardware Description Language
VMM	Verification Methodology Manual

Chapter 1

Introduction

During the last decades, electronic circuits have grown in complexity and in production costs which compelled engineers to research and develop new methods to verify the electronic design in more comprehensive, detailed and efficient ways.

The UVM methodology is one of the results of the increasing need of digital verification. It is designed in a way that allows to structure a verification environment in a reconfigurable architecture, so it can be possible to reuse components of the same environment across multiple technologies. This methodology is an industry standard recognized by Accellera System Initiative and it's comprised of a library for the SystemVerilog language (IEEE 1800) and a set of verification guidelines.

One of the main advantages in reusing components from different verification environments, consists in reducing the set up time of the verification of a new project, since it isn't needed to rebuild the reused components. Furthermore, devices that fall under the same category sometimes share similarities in their core architecture and, as a result, their verification environment can also share the same similarities. By studying these similarities, it is possible to develop a verification environment, focused in a special category of devices, which include the necessary features to be "ready to use", in a way that it is only required to add the components specific to each model while reusing the available infrastructure.

This project is targeted to a category of high-speed communication protocols developed by an EDA company, Synopsys. The purpose of this work is to take advantage of the best features of UVM and develop a reconfigurable verification environment that supports multiple communication protocols with minimal development effort. The project will start with the analysis of an existing verification environment used in a specific technology by Synopsys and then followed by an analysis of the verification techniques that could be used across different protocols.

So the goals defined for this project are:

- Analysis of an existing verification environment and removal of all design logic specific to the original protocol
- Revision of the verification environment in order to support multiple protocols

- Creation of generic blocks to support the revised environment
- Configuration and application of the generic environment to another existent protocol

From this disseration, it will result a well documentated verification environment, written in SystemVerilog and using the UVM methodology, that establishes an infrastructure which covers the core architecure of the high-speed communication protocols used by Synopsys.

During the development of the project, it was also created an UVM guide for beginners to this methodology. The guide includes a technical explanation of UVM and it is accompanied by a code example to serve as an example on how to build a complete verification environment with this methodology. It can be found in the appendix A of this document.

All the information about the project and the UVM guide can be consulted in the website built for this dissertation: http://colorlesscube.com/

1.1 Context

This dissertation was proposed by Synopsys Portugal and it was carried out as part of the Master's Degree in Electrical and Computer Engineering of the Faculdade de Engenharia da Universidade do Porto (FEUP).

The project was developed within the context of Synopsys' work with high-speed communication protocols and it is focused on devices that adopt these technologies.

Synopsys is one of the leading companies in the electronic design automation industry. Two of the most well known Synopsys' tools include Design Compiler, a logic synthesis tool, and VCS, a Verilog and SystemVerilog compiler. The later one was used as a main tool for this dissertation.

The offices of Synopsys Portugal in which this project took place are located at Maia and the the team behind the project was constituted by:

- The author of this document: Pedro Araujo
- Faculty Supervisor: Professor José Carlos Alves
- Company Supervisors: Luis Cruz & Domingos Terra

1.2 Structure of the document

This document presents the following structure:

- Chapter 2 provides some background regarding the subjects of hardware description languages, the need for hardware verification languages, the motivation behind verification methodologies and it also provides a technical overview of the Universal Verification Methodology.

- Chapter 3 presents a study of the devices targeted by this thesis and some situations that the verification environment has to support in order to fully verifiy this category of devices. In addition, a custom device was created in order to demonstrate the features of the environment.
- In chapter 4, the developed verification environment is described in detail, accompanied by the documentation of each class and the explanation of the design decisions taken during the conception of the project.
- Chapter 5 represents an application of the verification environment to the custom device created in chapter 3.
- Chapter 6 details the implementation of the same verification environment but to a different device, a model of the audio codec AC97
- Chapter 7, which is the final chapter of this document, presents a summary of the developed work throughout the semester and some conclusions.
- The appendix A presents the beginner's guide created during this dissertation.

Chapter 2

State of the art

Technology has advanced a long way and become increasingly complex. Its foundations started with computers whose logic was maintained by valves and that eventually moved to microscopic devices, like transistors.

In the early beginnings, electronic systems were designed directly at the transistor level by hand, but due to the increasingly complexity of electronic circuits since the 1970s [8], it became unpractical to design the core logic directly at the transistor level, so circuit designers started to develop new ways to describe circuit functionality independently of the electronic technology used. The result was the Hardware Description Languages and the era of Electronic Design Automation was born.

Hardware description languages are languages that are used to define the behavior and the structure of digital integrated circuits before they are translated into their own architecture. These kind of languages enable the modeling of a circuit for posterior simulation and, most importantly, translation into a lower level specification of physical electronic components.

A hardware description language resembles a typical programming language consists in a textual description of expressions and control structures and although they both share some similarities, they are not the same. One main difference is that HDL code is translated concurrently, which is required in order to mimic hardware, and while programming languages, after compilation, are translated into low level instructions for the CPU to interpret, HDL specifications are translate to digital hardware, so using hardware description languages requires a different mindset than using programming languages.

Nowadays, hardware description languages are the prevailing way of designing an integrated circuit, having superseded schematic capture programs in the early 1990s, and became the core of automated design environment. [10, p.15-1]

2.1 Hardware Description Languages

VHDL and Verilog are the two most popular HDL standards. [5] Verilog was created as a proprietary language by Phil Moorby and Prabhu Goel between 1983 and 1984 and it is formally based

in the C language. Cadence bought the rights of the language in 1989 and made it public in 1990. Eventually, IEEE adopted it as a standard in 1995 (IEEE 1364).

On the other hand, unlike Verilog which was originally designed to be used as a proprietary tool, VHDL was intentionally designed to be a standard HDL. It was originally developed on behalf of the U. S. Department of Defense between 1983 and 1984 but only released in 1985. It is based on Ada programming language and it was adopted as a standard IEEE 1076 in 1978. [10, p.15-3]

2.2 Functional Verification

Hardware description languages are tools that help engineers to easily specify abstract models of digital circuits to translate them into real hardware, but after the design is complete, another issue becomes noticeable: how can a designer know that the design works as intended?

This brings up the need for verification. Verification is defined as a process to demonstrate the functional correctness of a design. [4, p.1] This process is done by the means of a testbench, an abstract system that provides stimuli to the inputs of the design under verification (DUV) and analyses its behavior. A verification environment is represented in the figure 2.1.

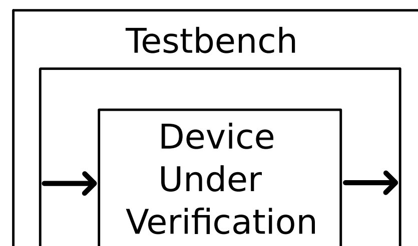

Figure 2.1: Generic testbench

One of the most common uses of a testbench is to show that a certain design implements the functionality defined in the specification. This task is known as functional verification. Normally, the testbench implements a model of the functionality that the designer wants to test and it is responsible to compare the results from that model with the results of the design under test. But it is important to take in account that functional verification can show that a design meets the specifications that have been verified but it cannot prove it. [4, p.2]

The traditional approach to verification relies on directed test. Verification engineers conceive a series of critical stimulus, apply them directly to the device under test (DUT) and check if the result is the expected one. This approach makes steady progress and produces quick initial results because it requires little effort for setting up the verification infrastructure. So given enough time, it maybe possible to write all the tests needed to cover 100% of the design. This scenario is represented in the figure 2.2.

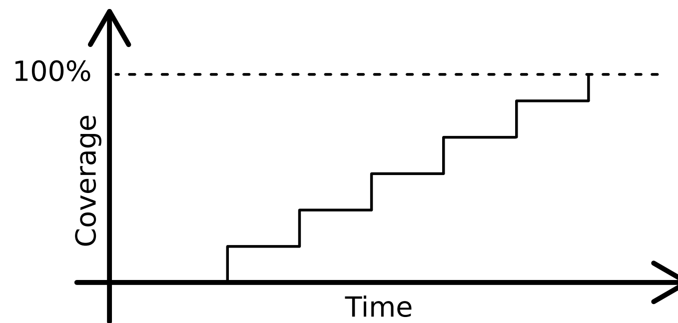

Figure 2.2: Direct testing progress [13, p.6]

But as the design grows in size and complexity, this becomes a tedious and a time consuming task. Most likely, there will be not enough time to cover all the tests needed in a reasonable amount of time and there will be bugs that the verification engineer won't be able to predict. So, random stimuli help to cover the unlikely cases.

However, in order to use random stimuli, there is the need of automating a process to generate them and there is also the need of a block that predicts, keeps track of the results and that analyses them: a scoreboard. Additionally, when using random stimulus, it will be needed to check what cases were covered by the generated stimuli, so the testbench will need functional coverage as well. Functional coverage is the process of measuring what space of inputs have been tested, what areas of the design have been reached and what states have been visited. [13, p.13]

This kind of testbench requires longer time to develop, causing an initial delay in the start of the verification process. However, random based testing can actually promote the verification of the design by covering cases not achieved with directed tests, as seen in the figure 2.3

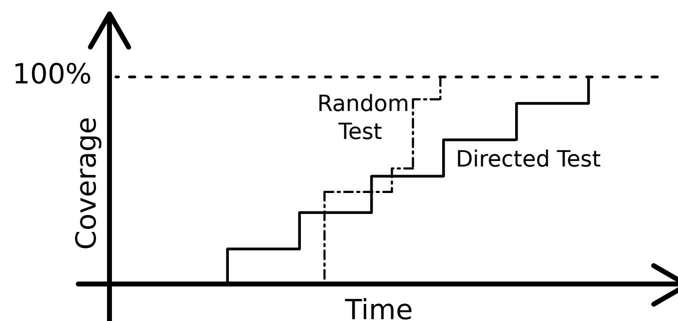

Figure 2.3: Random testing progress [13, p.8]

2.3 Hardware Verification Languages

In the previous section, it was mentioned some features that a verification environment must have: a stimulus generator, a functional coverage block, a scoreboard. However, it needs a block to drive the generated stimulus to the DUT and a block that listens to the communication bus, so that the responses of the DUT can be driven to the scoreboard block and to the functional coverage block. A representation of this structured testbench can be seen in the figure 2.4.

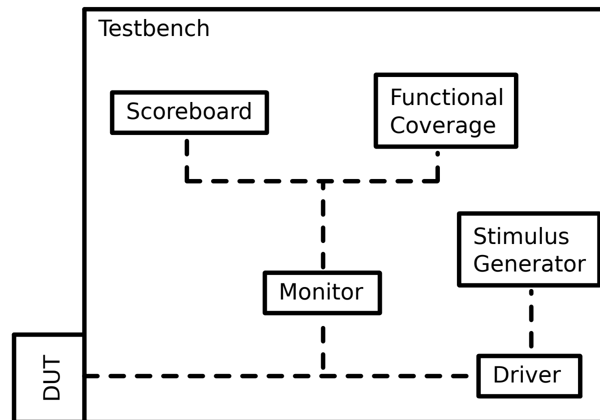

Figure 2.4: Structured testbench

As testbenchs grow and become more complex, the verification process might be a more resource consuming task than the design itself. Today, in the semiconductor industry, verification takes about 70% of the design effort and the number of verification engineers is twice the number of RTL designers in the same project. After a project is completed, the code of the testbench takes up to 80% of the total volume code. [4, p. 2]

As the circuit complexity grows, the verification process increases in complexity as well and become a very important and critical part of the project. However, the typical HDL aren't able to cope with the complexity of today's testbenches. Complex data structures, application of constraints to the random stimulus, presence of multiple functional blocks and functional coverage, are all examples of features that aren't available in the standard HDLs used for specification, namely in Verilog and in VHDL, as these more focused in creating a model of a digital design than programming verification. [7]

In order to solve this problem, Hardware Verification Languages (HVL) were created. The first one to be created was Vera in 1995. It was an object-oriented language, originally proprietary, designed for creating testbenchs for Verilog. The language was bought in 1998 by Synopsys and released to the public in 2001 as OpenVera. OpenVera has support for high-level data structures, constraint random variables and functional coverage, which can monitor variables and state transitions, and Synopsys also added support for assertions capabilities.

In 1997, a company named Verisity created the proprietary language *e*. Like OpenVera, it is also object-oriented and feature-wise, it is very similar to Vera.

IBM also developed its own verification language, Property Specification Language. It is more narrower than OpenVera or *e* and it is designed to exclusively specify temporal properties of the hardware design. [7]

Eventually, the features of some languages like, OpenVera and PSL, were merged to Verilog and it was created a new language, SystemVerilog. SystemVerilog supports a variety of operators and data structures, as well constrained random variables, functional coverage checking and temporal assertions. This new language was adopted as a standard IEEE 1364 in 2005 and is now the most used verification language in the industry. [10, p. 15-17]

2.4 Verification Methodologies

The adoption of verification languages eased the process of verification but there was no consensus in the proper use of a verification language. In the attempt of helping to deploy the right use of a verification methodology, Verisity Design (now Cadence Design Systems) published in 2000 a collection of the best practices for verification targeted to the users of *e* language. Named vAdvisor, it just consisted in a package of verification patterns in HTML format and it covered many aspects like coverage modeling, self-checking testbenches and stimuli creation. [9, p. xvii] In 2002, the same company revised vAdvisor and created the first verification library, the *e* Reuse Methodology (*e*RM). It was a big step because it included fundamental examples like sequences, objection mechanisms and scoreboards. [14]

In order to compete with Verisity Design, Synopsys presented in 2003 the Reuse Verification Methodology (RVM) for Vera verification language. The most notable contribution of RVM was the introduction of callbacks, that was inspired from software design patterns and adapted to verification languages. Eventually, RVM was ported from Vera to SystemVerilog to create the Verification Methodology Manual (VMM). [9, p. xviii]

Until this point in time, both verification methodologies had been proprietary and it was only in 2006 that it was introduced the first open source verification methodology, the Advanced Verification Methodology (AVM) from Mentor Graphics. [6] This library incorporated the concept of Transaction-Level Modeling from SystemC.

In 2005, after the acquisition of Verisity, Cadence started, as well, to port *e*RM to the standard of hardware verification languages, SystemVerilog. The result was the open source methodology, Universal Reuse Methodology (URM) in 2007.

However, in the joint task of merging the best features of each methodology, in 2008, Cadence and Mentor Graphics integrated both URM and AVM into a single open source methodology, the Open Verification Methodology (OVM). This collaborative effort ended up to be a good solution, because not only unified the libraries and the documentation but also, due to the open source nature, users could make their own contributions to the methodology. [9, p. xvii]

Later on, Accellera group decided to adopt a standard for verification methodologies and OVM was chosen as basis for this new standard and together, with Synopsys' VMM contributions, a new methodology was created: the Universal Verification Methodology (UVM). [1] A sum up of the evolution of verification methodologies is represented in figure 2.5.

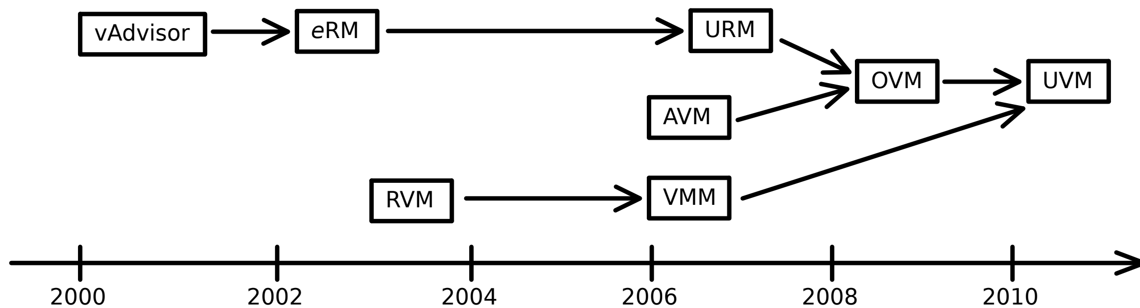

Figure 2.5: Evolution of verification methodologies

2.5 The Universal Verification Methodology

The UVM methodology is provided as an open-source library directly from the Accellera website and it should be compatible with any HDL simulator that supports SystemVerilog, which means it is highly portable. UVM is also based on the OVM library, this provides some background and maturity to the methodology. These two points are two of the main reasons for industry approval of the methodology.

Another key feature of UVM includes reusability. In a traditional testbench, if the DUT changes, engineers would redo the testbench completely. This process takes some effort but most of the times, if the testbench is correctly programmed, some blocks of it could be reused for the new testbench.

On other occasions, under the same DUT, verification engineers might want to apply a different test or change the kind of stimuli sent to the DUT. If the engineer doesn't take in mind that the test might change, he might end up revising the entire testbench. The lack of portability and documentation of the testbench might lead to a complete revision of the testbench without any margin for reusability.

UVM also addresses these kind of situations and specifies an API and guidelines for a standard verification environment. This way, the environment is understood by any verification engineer that understands the methodology and it becomes easily modifiable.

2.5.1 UVM Overview

The structure of an UVM environment is very similar to the testbenches mentioned previously. It features the most common components, like monitors, drivers and scoreboards, as well other classes that help to standardize testbenches across applications. A typical UVM environment is represented in the figure 2.6.

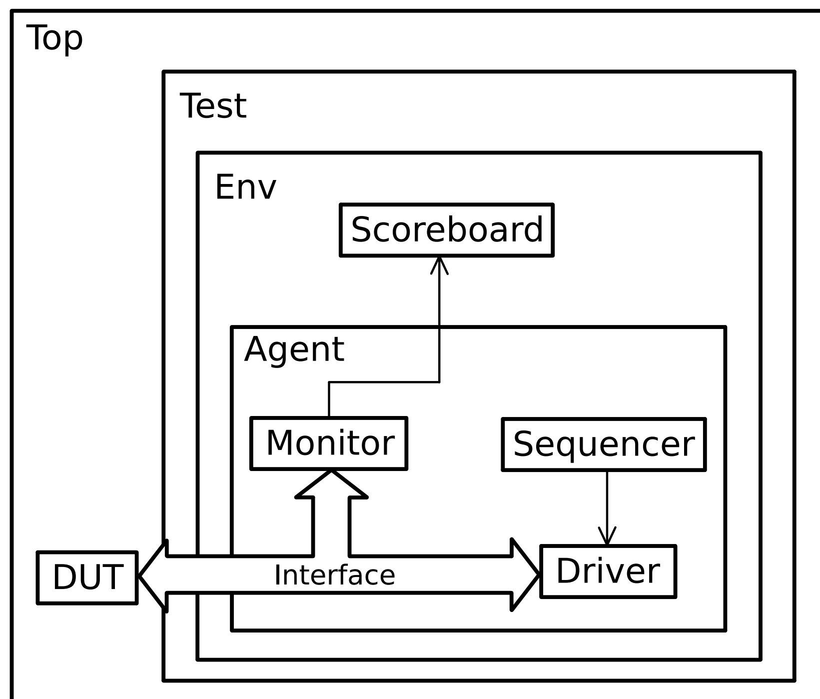

Figure 2.6: Typical UVM testbench

In the represented testbench, there is a device under test (DUT) and it is intended to interact with it in order to test its functionality, so there is the need to stimulate it. To achieve this, there will be a block, that generates sequences of data to be transmitted to the DUT, named *sequencer*. Information about the sequencer can be found in the appendix A.5

Usually sequencers are unaware of the details of the communication protocol, and are responsible for generating generic sequences of data to another block that handles the communication with the DUT. This block is called the *driver*. The appendix A.6 details the functionality of the driver.

While the driver maintains activity with the DUT, by feeding it data generated from the sequencers, it doesn't do any validation of the responses to the stimuli. The testbench needs another block that listens to the communication between the driver and the DUT and that evaluates the responses from the DUT. This block is the *monitor*. More information about the montor can be consulted in the appendix A.7.

Monitors sample the inputs and the outputs of the DUT, they try to make a prediction of the expected result and send the prediction and result of the DUT to another block, the *scoreboard*,

in order to be compared and evaluated. More information about the scoreboard can be read in the appendix A.9.

The components of the UVM environment communicate between each other by using the *Transaction Level* communication. This communication will be address later in sections 2.5.6 and 2.5.7

All these blocks constitute a typical system used for verification and it is the same structure used for UVM testbenches. Usually, sequencers, drivers and monitors compose an *agent*. An agent and a scoreboard compose an *environment*. All these blocks are controlled by a higher level block denominated *test*. The *test* block controls all the blocks and sub-blocks of the testbench. This means that just by changing a few lines of code, it is possible to add, remove and override blocks in the testbench and build different environments without rewriting the whole test.

To illustrate the advantage of this feature, imagine a situation where a DUT that uses SPI for communication is being tested. If, by any chance, we want to test a similar DUT but with I2C instead, it would just need to add a monitor and a driver for I2C, and override the existing SPI blocks, while keeping the sequencer and the scoreboard.

The current section gave an overview about the composition of an UVM environment. However, a deeper explanation of the UVM API will be provided in the following sections:

- The section 2.5.2 will explain the most important classes of the methodology
- The phases of each class will be described in section 2.5.3
- Each class has functionalities that are implemented by the usage of macros and the section 2.5.4 will explain the most important ones
- The section 2.5.5 illustrates the code for a generic UVM component
- The sections 2.5.6 and 2.5.7 refer to the Transaction Level communication between components

2.5.2 UVM Classes

The example from chapter 2.5.1 demonstrates one of the great advantages of UVM. It is very easy to replace components without having to modify the entire testbench, but it is also due to the concept of classes and objects from SystemVerilog. In UVM, all the mentioned blocks are represented as objects that are derived from the already existent classes.

A class tree of the most important UVM classes can be seen in the figure 2.7. [2]

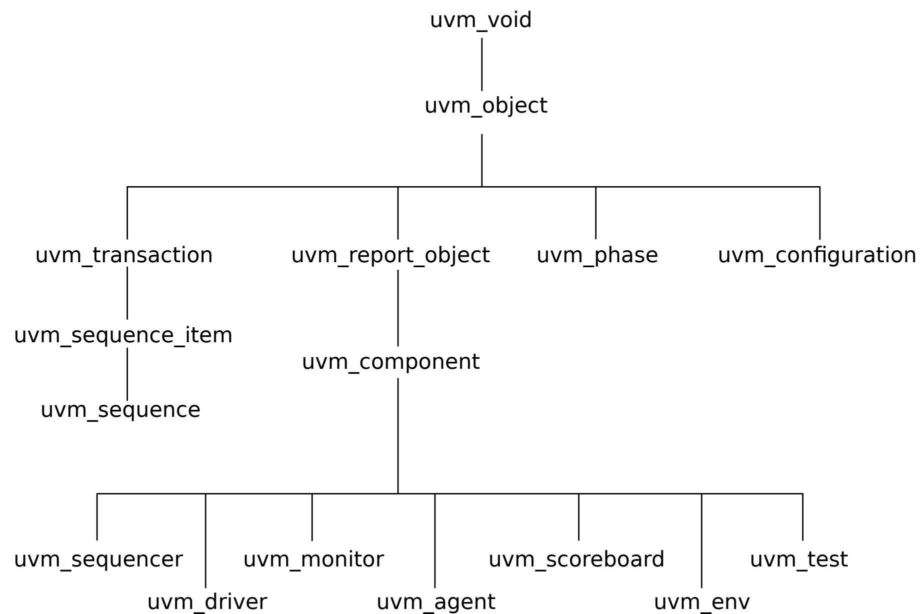

Figure 2.7: Partial UVM class tree

The data that travels to and from our DUT will stored in a class derived either from *uvm_sequence_item* or *uvm_sequence*. The sequencer will be derived from *uvm_sequencer*, the driver from *uvm_driver*, and so on.

Every each of these classes already have some useful methods implemented, so that the designer can only focus on the important part, which is the functional part of the class that will verify the design. These methods are detailed in the next sections.

2.5.3 UVM Phases

All the mentioned classes in chapter 2.5.2 have simulation phases. Phases are ordered steps of execution implemented as methods. When a new class is derived, the simulation of the testbench will go through these different steps in order to construct, configure and connect the testbench component hierarchy.

The most important phases are represented in figure 2.8. [1, p.48]

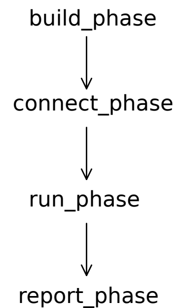

Figure 2.8: Partial list of UVM phases

- The build phase is used to construct components of the hierarchy. For example, the build phase of the agent class will construct the classes for the monitor, for the sequencer and for the driver.
- The connect is used to connect the different sub components of a class. Using the same example, the connect phase of the agent would connect the driver to the sequencer and it would connect the monitor to an external port.
- The run phase is the main phase of the execution, this is where the actual code of a simulation will execute.
- The report phase is the phase used to display the results of the simulation.

There are many more phases but none of them are mandatory. If a particular class is not needed, it is possible to omit it and the compiler will just ignore it.

2.5.4 UVM Macros

Another important aspect of UVM are the macros. These macros implement some useful methods in classes and in variables. They are optional, but recommended. The most common ones are:

- ‘uvm_component_utils - This macro registers the new class type. It is usually used when deriving new classes from *uvm_component*.
- ‘uvm_object_utils - This macro it is similar to the macro *‘uvm_component_utils* but it is used with classes derived from *uvm_object*.
- ‘uvm_field_int - This macro registers a variable in the UVM factory and implements some functions like copy(), compare() and print().
- ‘uvm_info - This a very useful macro to print messages from the UVM environment during simulation time.
- ‘uvm_error - This macro responsible for sending messages with an error tag to the output log.

These are the most used macros, their usage is the same for every class, but there are more macros available in the UVM API. [2, p. 405]

2.5.5 Typical UVM class

A typical UVM class has the structued listed in code 2.1.

```
class generic_component extends uvm_component;
	`uvm_component_utils(generic_component)

	//************************
	//** Variables
	//************************
	//The variables needed for the class go here

	//************************
	//** Constructor
	//************************
	function new(string name, uvm_component parent);
		super.new(name, parent);
	endfunction: new

	//************************
	//** Phases
	//************************
	function void build_phase(uvm_phase phase);
		super.build_phase(phase);
		//Code for constructors goes here
	endfunction: build_phase

	function void connect_phase(uvm_phase phase);
		super.connect_phase(phase);
		//Code for connecting components goes here
	endfunction: connect_phase

	task run_phase(uvm_phase phase);
		//Code for simulation goes here
	endtask: run_phase

	function void report_phase(uvm_phase phase);
		//Code for showing simulation results goes here
	endfunction: report_phase
endclass: generic_component
```

Code 2.1: Code for a generic component

The line 2 presents the macro for registering the component in the UVM environment, lines 11 to 13 represent the class constructor used to intialize the objects and variables from the class, and from line 18 is placed the functions and the tasks for the UVM phases described in section 2.5.3.

2.5.6 TLM-1: Ports

The first step in verifying a RTL design is defining what kind of data should be sent to the DUT. While the driver deals with signal activities at the bit level, it doesn't make sense to keep this level of abstraction as we move away from the DUT, so the concept of transaction was created.

A transaction is a class object, usually extended from *uvm_transaction* or *uvm_sequence_item* classes, which includes the information needed to model the communication between two or more components.

Transactions are the smallest data transfers that can be executed in a verification model. They can include variables, constraints and even methods for operating on themselves. Due to their high abstraction level, they aren't aware of the communication protocol between the components, so they can be reused and extended for different kind of tests if correctly programmed.

An example of a transaction could be an object that would model the communication bus of a master-slave topology. It could include two variables: the address of the device and the data to be transmitted to that device. The transaction would randomize these two variables and the verification environment would make sure that the variables would assume all possible and valid values to cover all combinations.

Now, another question arises: how are transactions transported between components? The answer is: through TLM. Transaction Level Modeling and it's a high-level approach to modeling communication between digital systems. TLM provides a set of communication interfaces that can be used to connect different components at the transaction level by isolating them in the environment. This promotes reusable components and minimizes the time required to build a verification environment. [3, p. 9]

There are two kinds of TLM communication interfaces: TLM-1 and TLM-2.0. This chapter will focus on TLM-1 but more information about TLM-2.0 can be consulted later on chapter 2.5.7.

The TLM-1 is represented by two main aspects: *ports* and *exports*. A TLM port defines a set of methods and functions to be used for a particular connection, while an export supplies the implementation of those methods. Ports and exports use transaction objects as arguments.

Figure A.12 is possible to see a representation of a TLM-1 connection.

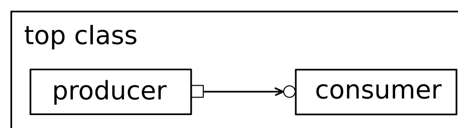

Figure 2.9: Port-export communication

The consumer implements a function that accepts a transaction as an argument and the producer calls that very function while passing the expected transaction as argument. The top block connects the producer to the consumer.

A sample code is provided in table 2.1.

Table 2.1: Sample code for ports and exports

```
class topclass extends uvm_component;
        ...
        function void connect_phase(uvm_phase phase);
                ...
                producer.test_port.connect(consumer.test_export);
                ...
        endfunction: connect_phase
endclass: topclass
```

```
class producer extends uvm_component;
uvm_blocking_put_port#(test_transaction)
        test_port;

        ...

        task run();
                test_transaction t;
                test_port.testfunc(t);
        endtask: run
endclass: producer
```

```
class consumer extends uvm_component;
uvm_blocking_put_imp#(test_transaction, consumer)
        test_export;

        ...

        task testfunc(test_transaction t);
                //Code for testfunc() here...
        endtask: testfunc
endclass: consumer
```

The class *topclass* connects the producer's *test_port* to the consumer's *test_export* using the *connect()* method. Then, the producer executes the consumer's function *testfunc()* through *test_port*.

A particular characteristic of this kind of communication is that a port can only be connected to a single export. But there are cases when we might be interested in having a special port that can be plugged into several exports.

A third type of TLM port, analysis port, exists to cover these kind of cases.

An analysis port works exactly like a normal port but it can detect the number of exports that are connected to it and every time a required function is asked through this port, all other components whose exports are connected to an analysis port are going to be triggered.

Figure 2.10 represents an analysis port communication.

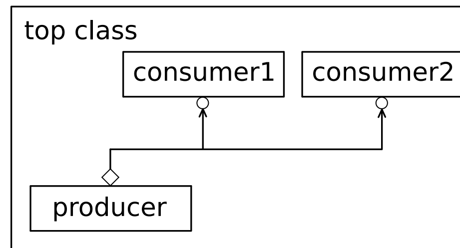

Figure 2.10: Analysis port communication

The usage of analysis ports is very similar to normal ports, so it is not provided a code example like in the table 2.1. However, a brief summary of these ports and exports can be seen in the table 2.2.

Table 2.2: Sum up of TLM-1.0 ports

Symbol	Type	Port declaration
□	Port	uvm_blocking_put_port #(transaction) port_name
○	Export	uvm_blocking_put_imp #(transaction, classname) export_name
◇	Analysis Port	uvm_analysis_port #(transaction) analysis_port_name

2.5.7 TLM-2.0: Sockets

In chapter 2.5.6, TLM-1 ports were analyzed but they have two major advantages: they don't support bi-directional communication and they don't have passthrough objects. To solve these problems, a new kind of connection was created: TLM-2.0, most known as *sockets*. A socket is very similiar to a port or an export, as it is derived from the same class, but it provides a forward and a backward path. [3, p. 24]

A socket connection is starts with a component that has *initiator sockets* and ends with another component that has *target sockets*. Initiator sockets can only connect to target sockets and vice-versa and each socket can only have a maximum of one connection. The figure 2.11 represents a typical socket connection between a target and a initiator.

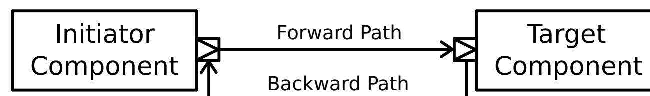

Figure 2.11: Representation of a socket communication with an initiator and a target component

It is also possible to implement passthrough socket in case it is desirable to relay the message from a component to a subcomponent. This situation is represented on figure 2.12.

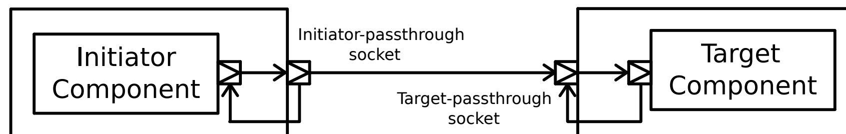

Figure 2.12: Socket communication with an initiator, a passthrough and a target component

These two cases represent the situations that will be using during the execution of this project.

2.6 Conclusion

Verification plays a big role in the conception of digital integrated circuits. Manual test cases are not enough anymore to do a thorough verification of a digital design, so verification languages and verification methodologies were created in order to assist engineers to test even the more unlikely cases.

The current chapter presented an analysis to functional verification and to verification methodologies, and presented the most essential material used for the development of this thesis. A more extensive explanation of UVM can be found in the Appendix A, this should provide an extra explanation of some concepts talked over the next chapters.

Chapter 3

Analysis of communication protocols

Nowadays, electronic consumer devices are more popular than ever, it's a trend in constant growing and in great market need. One of the most popular electronic devices are the smartphones.A smartphone is a small, battery operated, highly sophisticated computing and communication device. These are actually, one of the major market drivers for low power consumption and high performance devices. So hardware designers need to take these aspects in mind during the development of the hardware.

One of the aspects that can be improved in electronic devices in order to reduce its power consumption it's the efficiency of the interconnection between their internal components, for example, between a camera and the CPU or the modem and the antenna. An ideal system-on-a-chip must incorporate multiple high speed peripherals for inter-chip communication, while maintaining a low power consumption. In response to this need, there have been many developments to optimize communication interfaces at the physical layer for mobile applications.

3.1 X-PHY Overview

This thesis will focus on technologies developed by Synopsys for high-speed communication protocols. As the specific details of these technologies must remain confidental, we will use a generic term demoninated of *X-PHY* to refer them.

The physical layer, commonly known as PHY, is the lowest layer of the OSI model. It is the bridge between the physical medium and the link layer, so it is the ideal place to optimize power consumption. A PHY serves as the base for other protocols at the higher layers that deal with other purposes, such as data storage, data transfer, interaction with displays, etc. An optimized physical layer for mobile applications, let's assume X-PHY, could replace some of the communication interfaces of other technologies in order to reduce the power consumption while preserving the characteristics of that technology. The X-PHY is considered as a serial communication interface in which the data is sent through differential pins.

An example of the uses of X-PHY can be seen in the figure 3.1.

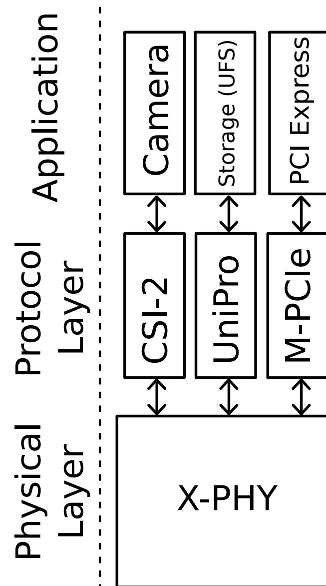

Figure 3.1: Applications of a possible X-PHY

In order to support a technology like PCI-Express on a mobile environment, a protocol layer, like the M-PCIe, can be implemented on top of X-PHY. The usage of X-PHY as the physical layer instead of the PCI Express PHY, could mean optimization in power consumption for mobile applications while still providing a high-speed interface.

Another advantage of X-PHY besides an optimized power consumption, is the low pin count. The most basic unit of the X-PHY is the lane, which can be a transmitter (TX) or a receiver (RX), and each lane it is represented with just a pair of differential pins. This can be seen on the figure 3.2.

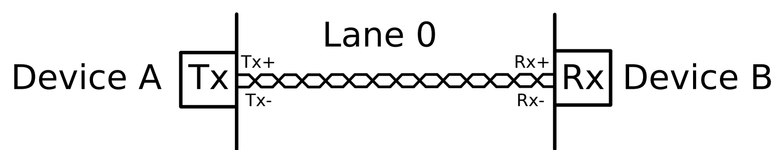

Figure 3.2: X-PHY basic lane

As a consequence of preserving power, each lane supports multiple states. These states represent a trade-off between power and performance and can be dynamically adapted to the situation's need. The states of X-PHY are represented in figure 3.3.

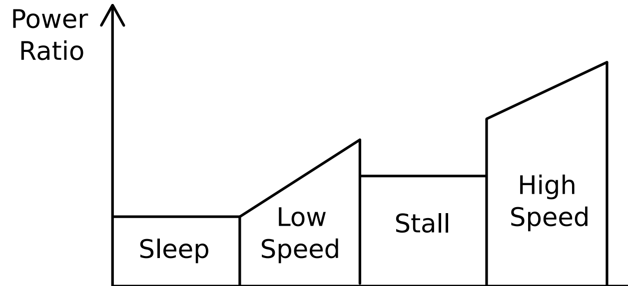

Figure 3.3: States of X-PHY

For data transmission, there are two main states, *Low-Speed* (LS) and *High-Speed* (HS). This availability of multiple power modes, allows the designer to have some versability on the approach to the relation between the best performance and power consumption. This means that X-PHY allows for situations where it is demanded a constant high-speed communication without caring for power saving modes, as well for situations where it is important to preserve power.

The figure 3.4 represents 3 different situations that trade-off latency in communication with power saving modes.

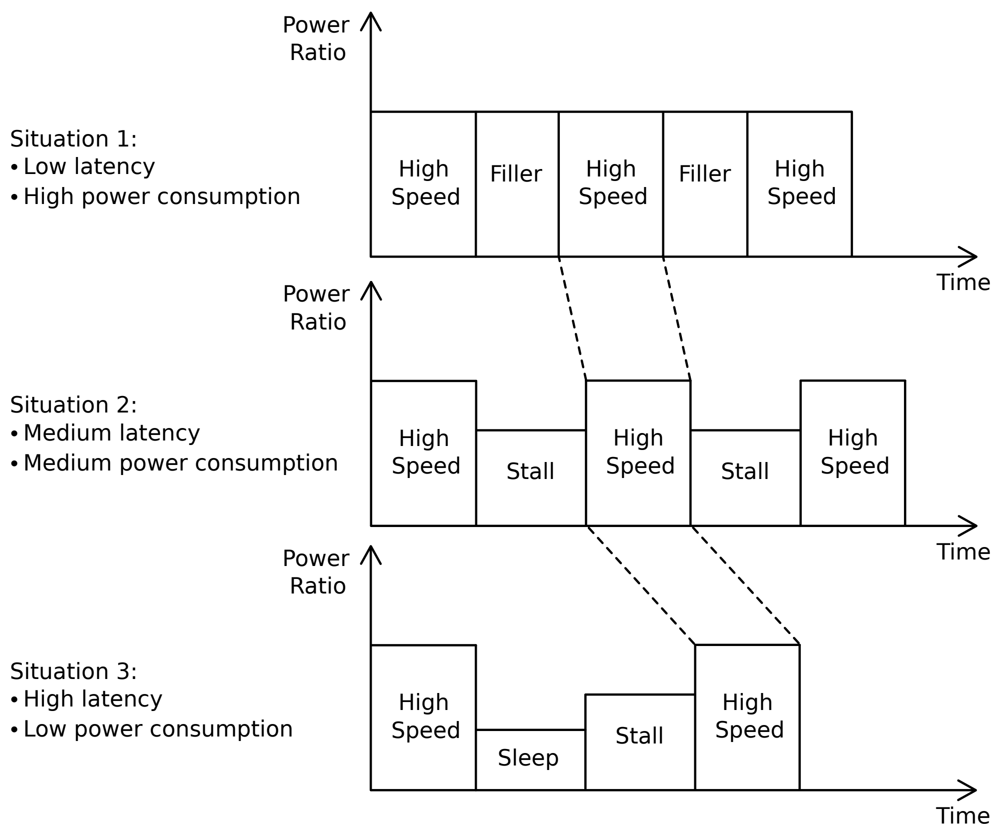

Figure 3.4: Representation of 3 different situations for a trade-off between power and performance

The situation 1 represents a situation where a constant high-speed communication is important without taking in account power saving features. In this scenario, the interface fills the transmission with filler data in order to keep the high speed states even during periods of no data transmission, making the communication more responsive. As a drawback, this causes a higher power consumption.

To reduce it, instead of filling the gaps with filler data, the X-PHY can enter in a *stall* state that saves up more power. The change of states, causes a small latency between the transmissions of frames and thus making the communication less responsive. This is represented by situation 2.

The situation 3 represents a case even more optimized for power. After a frame has been transmitted in high-speed, the interface goes into the *sleep* state, a state that can save more power than *stall*. This introduces a higher latency because the interface, before going into the high-speed state again, has to leave the *sleep* state and go through the *stall* state.

All these cases refer to operations with just one lane but X-PHY also supports multiple lanes simultaneously. The operation mode with just the basic lane configuration states the basic functionality of X-PHY and the configurations with multiple lanes add a common block shared by all of them that manages all the lanes' states, this block is denominated of *Lane manager*.

The figure 3.5 represents a possible situation of X-PHY with more than just one lane.

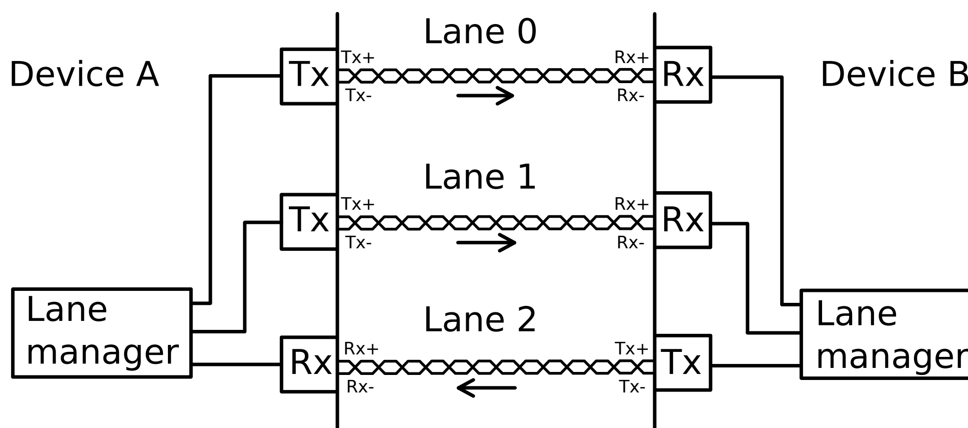

Figure 3.5: X-PHY with 3 lanes

Another possible scenario would be 3 transmitters and 3 receivers on the same device, or even just 4 receivers. Although all these lanes don't share the same bus, they are dependent of each other. For example, there may exist cases when the Lane 1 be must inactive while Lane 2 is transmitting, or when Lane 3 is going to high-speed and all other low-speed lanes must switch to high-speed too. This behavior is controlled by the lane manager.

All these situations were mentioned in order to understand the kind of configurations that a verification environment has to support. These include: single lanes supporting multiple states; multiple lanes being dependent of the states of other lanes; add or remove lanes dinamically in order to test the behavior of the PHY with multiple configurations. An I2C or SPI interface are much simpler and don't need all this kind of complexity. An ideal verification environment would provide the necessary infrastructure to deal with the described situations.

3.2 X-PHY Verification

The main functionality of X-PHY was analyzed and with that analysis it is possible to take an approach of a possible verification environment. At its essential level, a verification environment emulates the functionality of the device under testing (DUT) and compares the result with the DUT's behavior.

Taking into account the figure the 3.2, the *Device B* (the receiver part) is considered as a DUT for the situation that will follow. A testbench for this DUT, would need a component that stimulates the device, a *driver*, and it would need a component that listens to the serial line in order to emulate the functionality that we want to test, a *monitor*. It would also need another *monitor* that would watch over the DUT's behavior and a component that would compare the result from the DUT and from the testbench, a *scoreboard*.

Testing a simple configuration like this is quite simple. The X-PHY is a communication protocol, so the basic functionality that the testbench needs to verify is if the data got correctly interpreted by the device. The testbench would compare the data obtained through the monitor watching the serial line with the data interpreted by the DUT and it would look for any mismatches.

Figure 3.6 represents an UVM environment that can be used to build the described testbench.

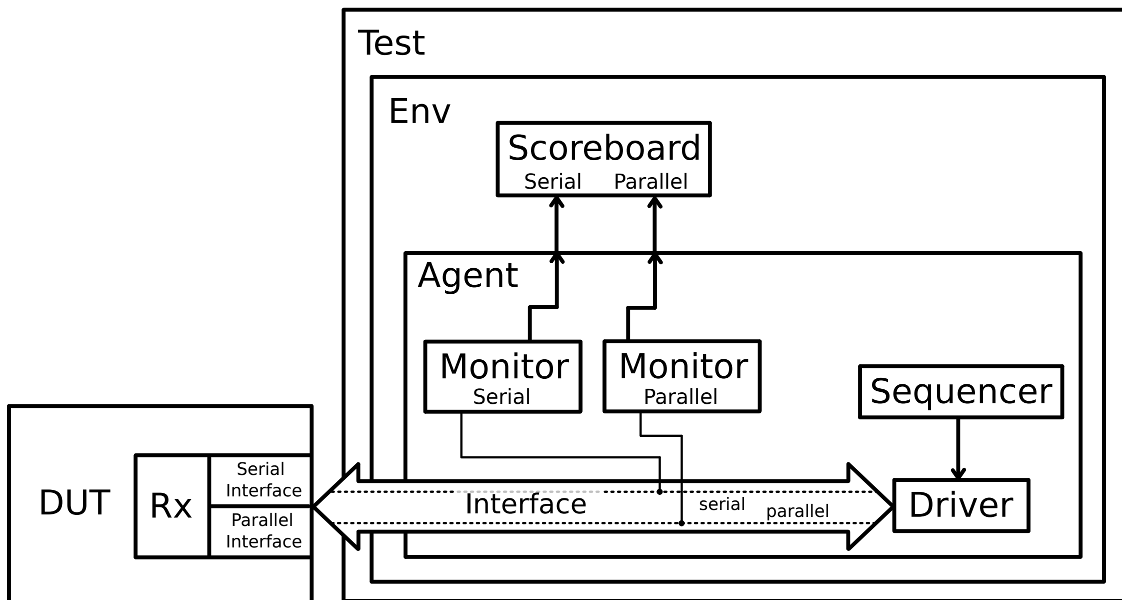

Figure 3.6: An UVM verification environment for the receiver of X-PHY

As we are testing the Rx interface, the testbench will emulate the Tx side of the communication. This analysis to the verification of X-PHY will focus mostly on the receiver part, but the process is the same for the trasmitter.

The *sequencer* is responsible for feeding transactions to the driver and it shouldn't be connected to any other block except to the driver, because only the monitors are responsible for the evaluation of the DUT's functionality, although the sequencer can hold information about the type

of transactions that should be sent to the environment.

The *UVM User's Guide* from Accellera [3, p.4] describes the *agent* as an abstract container, that encapsulates a driver, a sequencer and a monitor, with the purpose of emulating the DUT. These three blocks act independently in the testbench. And by putting them in a container, it becomes easier to reuse them for other situations.

In order to demonstrate reusability, it is going to be assumed that the DUT now features two similar, but independent, Rx interfaces. In order to verify the functionality of the second Rx interface, we just need to duplicate the *agent* and the *scoreboard*. The figure 3.7 represents this scenario.

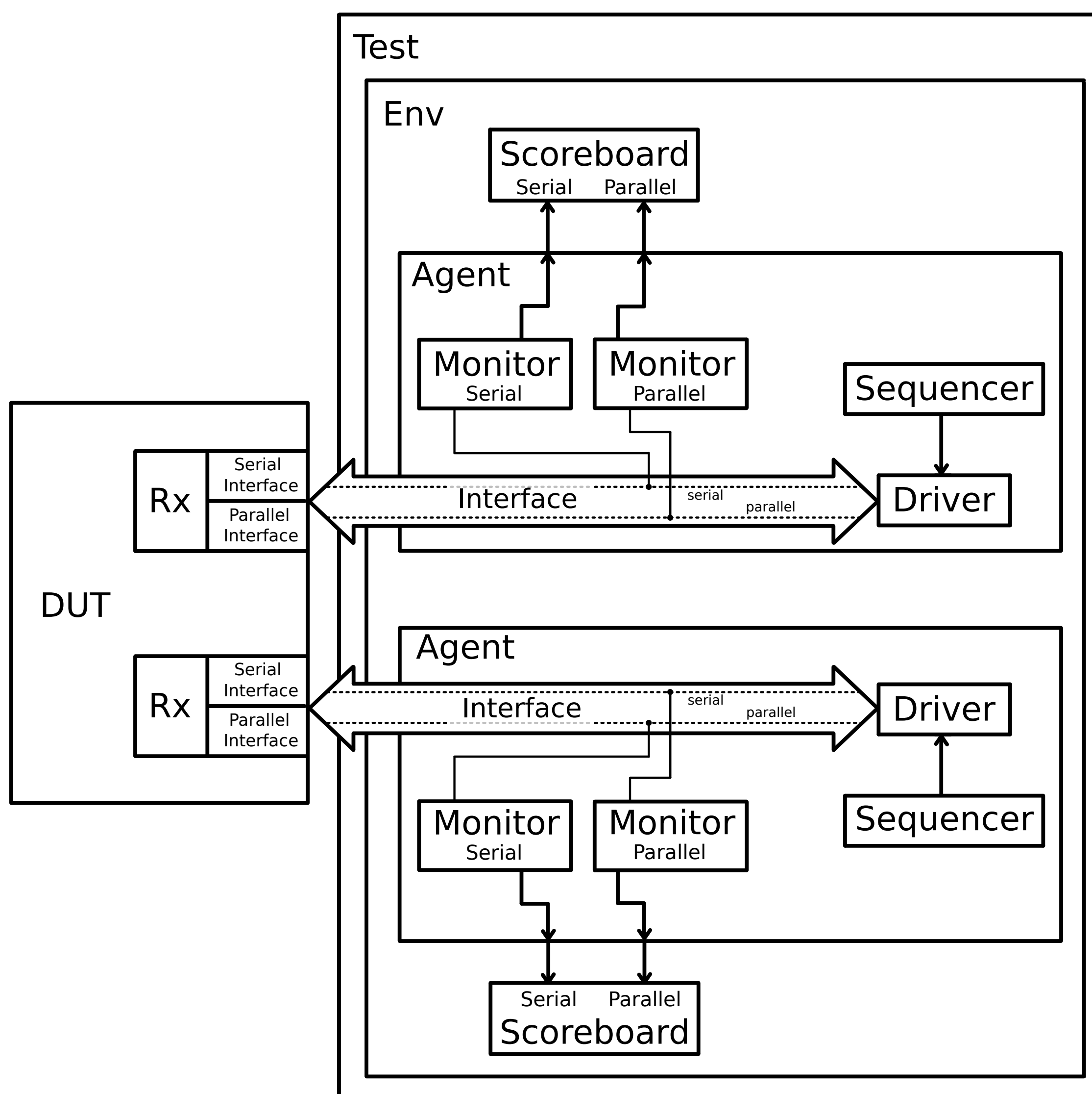

Figure 3.7: An UVM testbench with 2 instances of the same agent for a X-PHY device with 2 Rx

It is possible to notice that for this situation, the *scoreboard* is always closely related to the *agent* so it would make sense to put it together with the monitors, the driver and the sequencer.

It is now considered different DUTs: a DUT with one Rx interface (figure 3.6) and another DUT with two Rx interfaces (figure 3.7).

A typical testbench would keep two separate *Env* blocks, an Env with only one agent and another Env with two agents, but this would mean managing two separate tests. The ideal testbench would be able to adapt itself to these two situations without reworking the whole environment. To achieve this, each agent will feature a configuration block that will enable or disable the respective agent depending on the device being tested.

This new verification environment is represented on figure 3.8.

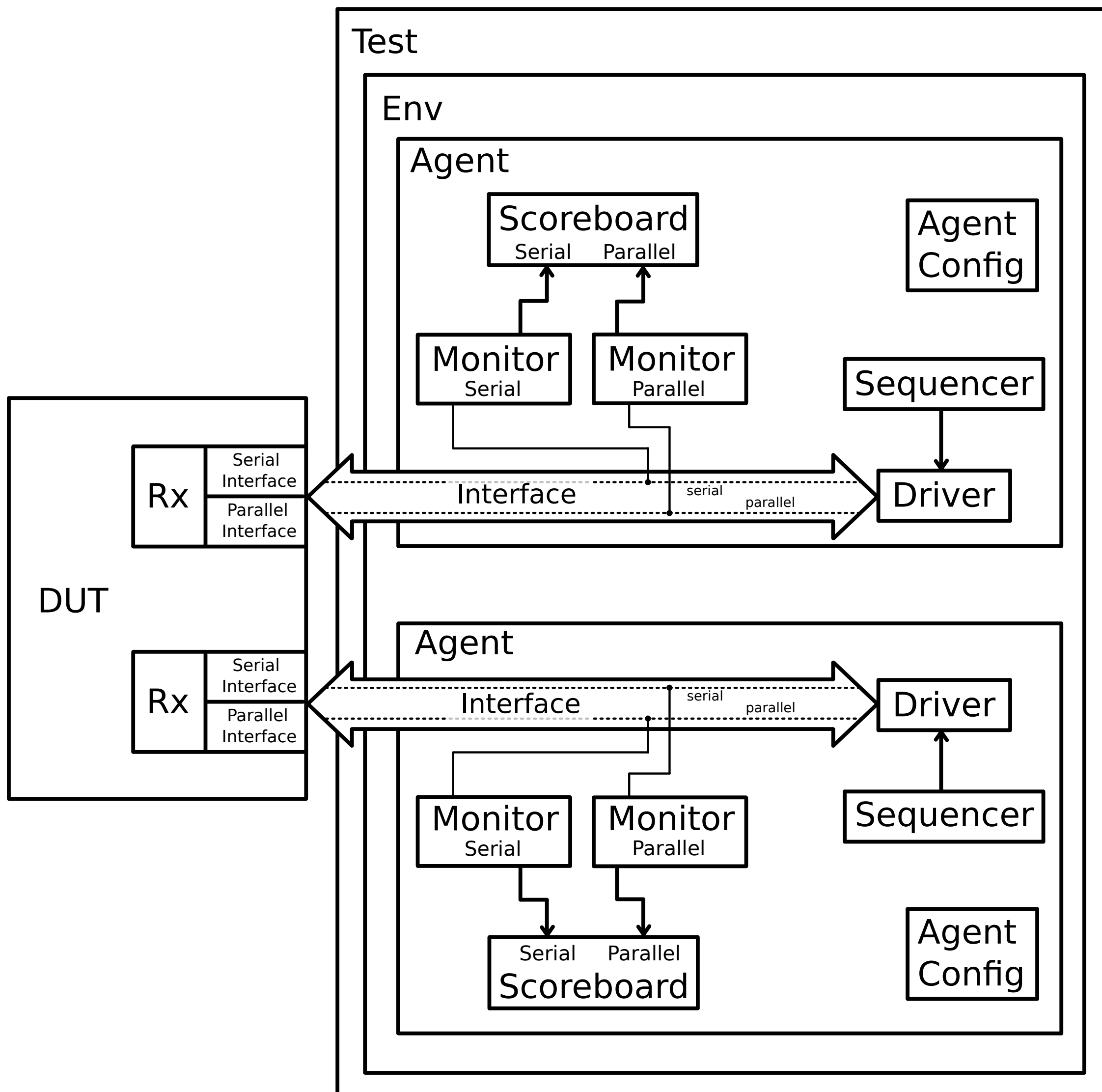

Figure 3.8: An UVM verification environment with support for agent configuration

The new *agent config* block will allow for a reconfigurable verification environment and with just one line of code, in the *test* class, it will be possible to disable and enable agents manually. This is useful for verifying DUTs like X-PHY that can have different configurations, as seen in figures 3.2 and 3.5.

Each agent now is composed by a driver, a sequencer, two monitors, a scoreboard and a configuration block. This is enough to test a simple interface individually: the sequencer generates the necessary sequences to be sent to the DUT and the driver takes cares of the communication with the device by emulating the X-PHY's states and trasmission frames. Meanwhile, the monitors take care of collecting transactions from the communication to be evaluated by the scoreboard.

But if it is intended to test a DUT in which the lanes depend on each other, the testbench will have to support a block that manages multiple agents at the same time in order to emulate the same functionality. This is represented on figure 3.9.

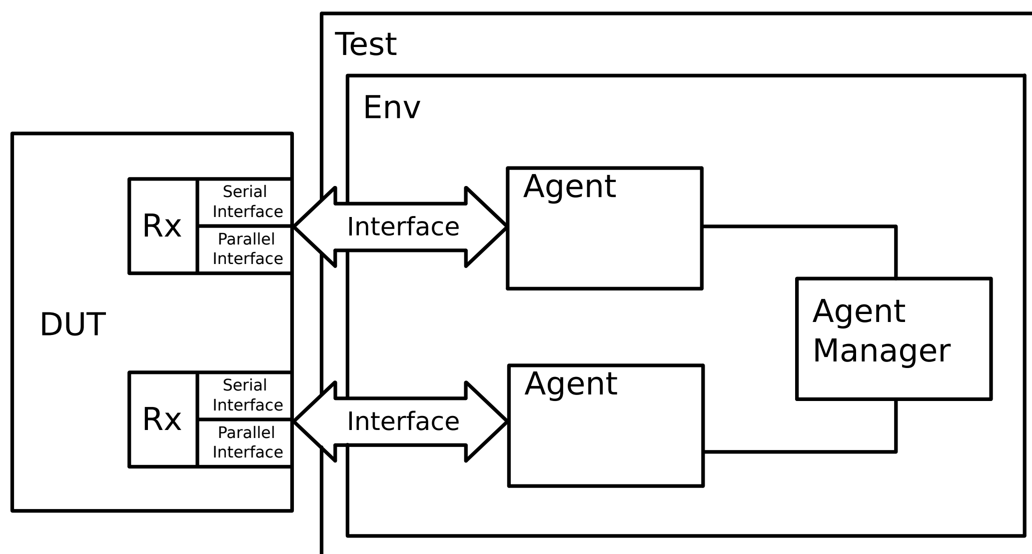

Figure 3.9: An UVM verification environment with support for an agent manager

The *agent manager* will be connected to the drivers and to the monitors of each agent and it will consist of a state machine that controls transmission requests from all the lanes.

As an example, assuming a hypothetical situation that in the DUT there is a resource shared by both interfaces, like a clock generator, and assuming that the change of states of each lane would depend on the system clock, the lane manager would control the access to this shared resource, thus controlling the change of the states of each lane.

This would mean that if both lanes are transmitting in the high-speed state and one of them would want to change it is state to low-speed, this lane would first make a request to the lane manager and the lane manager in its turn would check for the usage of the shared resource and give permission, or not, for the lane to change its state.

The envisioned testbench would support for multiple configurations just by changing a few lines of code. The presence of an *agent manager* also allows for a dynamic reconfiguration of the testbench during the execution of the test.

In order to demonstrate some of these features, it will be developed a device that will emulate the features of X-PHY. For this device, it will be used an I2C interface instead of the interface X-PHY. The following section will start with a brief overview of the I2C protocol before presenting the example device.

3.3 I2C Overview

The I2C is a synchronous communication protocol originally developed by Philips during the 1980s as a way to establish communication between several electronic devices, like microcontrollers, displays and sensors. It features a serial bus interface with a master-slave topology on top of two physical wires. It is capable to reach speeds up to 100 Kbps (Standard-mode) and up to 400 Kbps (Fast-mode). [12, p.3]

The only physical lines available in a typical I2C communication are the serial data line (*SDA*) and the serial clock line (*SCL*) and all I2C devices from the same network share same two lines. The clock line is always generated from the master but the data line can be controller either by the master or by the slave. A typical I2C topology is represented on figure 3.10.

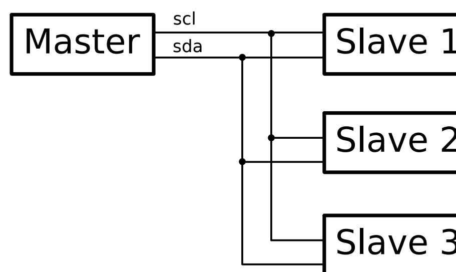

Figure 3.10: Typical topology for an I2C interface

The communication on an I2C bus is always started by the master device which addresses to the desired slave by an unique slave address. The I2C address has 7 bits and the maximum number of nodes in the same network is thus limited to 127 different devices.

The master starts the communication by issuing a *start condition* and ends the communication with a *stop condition*. The start condition occurs with the transition from high to low of the SDA while the SCL line is in high state and the stop condition occurs with the transition from low to high while the SCL line is pulled high. These conditions are represented on figure 3.11.

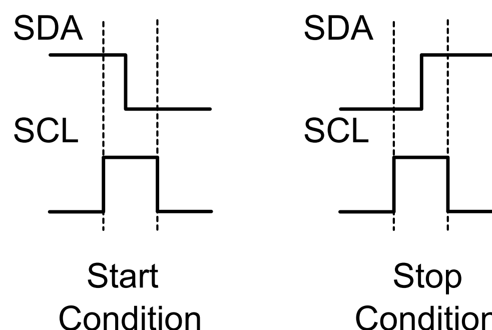

Figure 3.11: Representation of a start and stop condition on an I2C bus

After the start condition, the master transmits the address of the slave and the read/write bit, depending on the type of operation that master desires to do, and waits for an acknowlegment of the slave. If there is a slave with the transmitted address listening to the bus, the slave will acknowledge the request and read, or write, a byte, from the bus and another acknowledge will follow. The communication finally ends with the stop condition from the master.

A representation of a read operation (the R/W bit is 1) is shown in the figure 3.12. The top of the figure represents the action of the master in the bus and on the bottom the action of the slave.

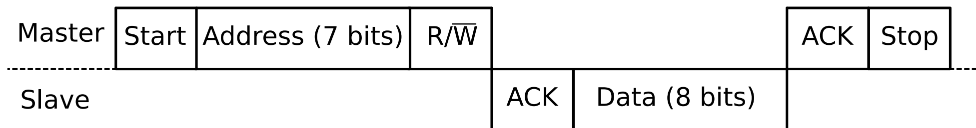

Figure 3.12: Representation of an I2C read operation

It is possible to note that in this frame it is the master that acknowledges the data. In the I2C protocol, the communication is done in blocks of 8 bits and the node that receives those 8 bits is the one that send the acknowledgement to the bus. The first block of 8 bits is always sent by the master, which is the slave address of 7 bits plus the read/write bit. The second block can be either a write operation to a slave or a read operation from a slave.

So in figure 3.12 the master requests data to be read from the slave, by setting the bit R/W to 1, and the slave sends a block of 8 bits of data. The master then confirms the reception of the block by sending an ackowledgement signal.

Figure 3.13 represents a write operation (the R/W bit is 1). In this case, the master sends 8 bits of data to the slave and the slave has to confirm the reception of the data, so it sends an aknowledgement signal.

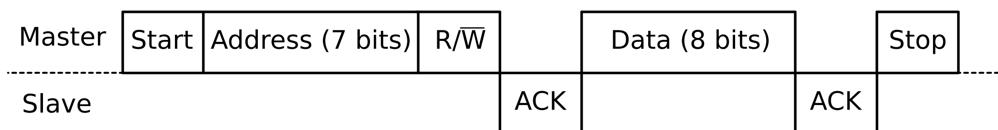

Figure 3.13: Representation of an I2C write operation

It is also possible to transmit more than one byte of data in just one I2C frame. After the ACK of the first data byte, instead of generating a *stop condition* the master just needs to send the next byte, in case of the write operation, or wait for the next byte from the slave, in case of read operation. This situation is represented in figure 3.14.

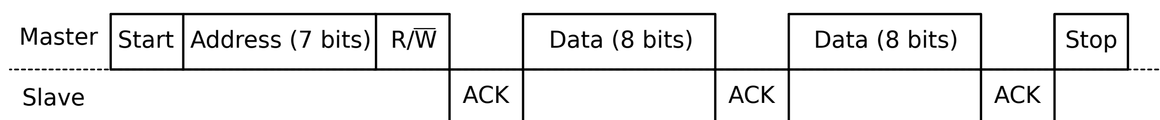

Figure 3.14: Representation of an I2C write operation of 2 bytes

If by any chance, in the beginning of the communication the send address isn't recognized by any slave (if there isn't any acknowledgement in the line), the master assumes that there was no communication and just retries again as it is represented in figure 3.15.

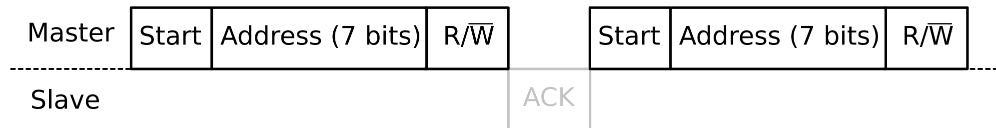

Figure 3.15: Representation of an I2C write operation without slave acknowledgement

As an example, figure 3.16 shows a timing diagram for a typical I2C communication. The address sent is 7'b1010110 and the master is accessing the slave in write mode (R/W is 1'b0). The data received from the slave is 8'b11010010.

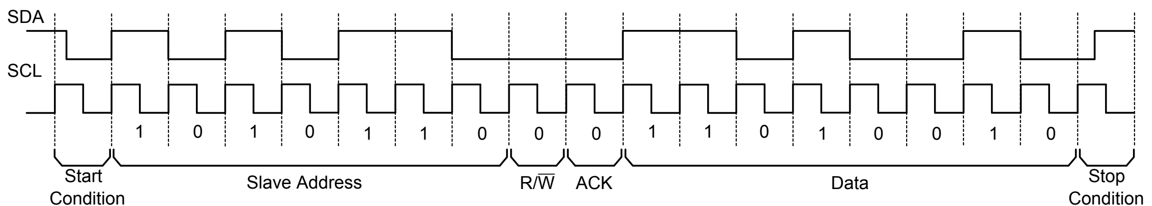

Figure 3.16: Representation of a typical I2C timing diagram

The next chapter will explain how a verification of a single I2C could be done.

3.4 I2C Verification

After studying how an I2C interface works, it is possible to draft an early verification plan for it. This chapter will go through a possible verification plan for a single I2C slave and a single I2C master and this plan will be used as a base model for the device created in section 3.5. This section will mostly focus on the basic components needed for building a verification environment and a summary of these components can be found by the end of the section.

In order to assist the verification process, a parallel interface will be added to the I2C devices. This interface will permit to access the internal registers of the DUT, allowing for the testbench to make a proper validation of the DUT's operation. The pinouts of this interface can be inputs and outputs at the same time.

The interfaces available for each I2C device are represented on the figure 3.17.

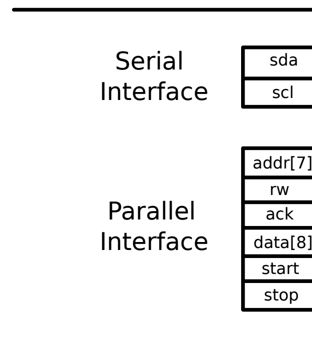

Figure 3.17: Serial and parallel interfaces created for the I2C device

The testbench for the slave interface will act as a master and the testbench for the master interface will act as a slave, this means that it will be needed a driver that acts like as an I2C master and another driver that acts as an I2C slave.

To listen to the communication bus, it will be needed a monitor that acts as a slave for both cases. Another monitor to listen to the interface's registers will be required and also a scoreboard to compare the results from both monitors. These two components can be reused in both testbenches as well.

There are three main parameters on an I2C communication: the slave address, the read/write bit and the data. These three parameters will act as a starting point for the verification plan.

It will first be considered a slave I2C interface and then the document will move to the verification of an I2C master.

3.4.1 Verifying an I2C slave

There are four main situations it is desirable to test:

- The slave has to acknowledge only to his address and completely ignore all the communications with other addresses
- The slave has to write into the bus when the read/write bit is equal to 1'b0 and read the bus when the same bit is qual to 1'b1
- The slave cannot write into the bus unless at master's request
- The data sent to the slave and sent by the slave, has to be correctly interpreted

Using these tests and the parameters mentioned previously, it was created a transaction that will model the communication between the testbench and the DUT.

The transaction is named *i2c_trans* and it is a class derived from *uvm_sequence_item*. The structure of the transaction can be seen on the table 3.1.

Table 3.1: I2C transaction

Transaction: i2c_trans		
Name	Description	Variable
Slave address	The address to be sent in the I2C frame	logic [6:0] slave_address
Read/write bit	The bit that defines the type of operation	logic rw
Data	The 8 bit message sent through the bus	logic [7:0] data
Number of retries	The number of times that the master tries to communication in case the lack of ACK	int n_retries

This transaction is used by a driver that will build an I2C frame and interact with the DUT. The driver is connected to a sequencer, named *i2c_sequencer*, which, on its turn, will take transactions from another block, the *i2c_sequence*, and then pass along random transactions to the driver. More information about the relationship between the sequences and the transactions can be consulted in appendix A.5.

The driver will act as an I2C master device and it will belong to the class *i2c_driver_serialmaster*. The run phase will go through these events:

1. Retrieve a random sequence from the sequencer
2. Set a start condition
3. Transmit the random address from the sequencer and the read/write bit
4. Wait for an acknowledgement for the slave
5. If there is an acknowledgement, it will transmit the data. If not, if not, it will retry the number of times defined in the transaction
6. Set a stop condition
7. Report to the sequencer that the operation with the transaction is finished and repeat the process

The slave address of each I2C slave interface is defined by the port *addr* of the parallel interface. This port is accessed by a component in the testbench, usually the driver, that obtains the address through a configuration block.

The sequencer generates random transactions for the driver but there are cases when it's desirable to force certain tests to the interface. The ideal class to define the values to be used on tests is the *test* class. In order to pass values from the *test* class to the sequencer, a configuration block will be created, the *i2c_agent_config*. This configuration block will be submitted to the UVM configuration database, to be later retrieved to the sequencer.

The composition of the configuration block is very similar to the transaction and it will hold values to force a type of test to the DUT. It is a class named *i2c_config* and it is represented in table 3.2.

Table 3.2: I2C Agent Config

Agent config: i2c_config		
Name	Description	Variable
Slave address	The address to be sent in the I2C frame	logic [6:0] slave_address
Read/write bit	The bit that defines the type of operation	logic rw
Data	The 8 bit message sent through the bus	logic [7:0] data
Number of retries	The number of times that the master tries to communication in case of lack of ACK	int n_retries
Actual slave address	The actual address of the slave being tested. This address is sent to the port *addr* of the parallel interface	logic [6:0] real_address
Agent active	Variable that defines if the bit is active or not	bit active_agent

And while the driver maintains activity on the bus, there will be a component listening to the same activity. This component will be reacting to the driver's action on the bus by emulating the DUT's behavior. This component will a monitor and it will belong to the class *i2c_monitor_serialslave*. So it will go through these events:

1. Wait for a start condition
2. Collect the address and the read/write bit
3. If the address is a match, internally acknowledge the information (but it doesn't change anything on the bus), otherwise discard any further information
4. If the address was a match, store the 8 bit data traveled through bus into an empty transaction
5. Wait for a stop condition
6. Send the collected information to the scoreboard and repeat the process

There will also be a monitor that will be listening to the parallel interface of the device checking how the DUT reacts to the driver: a monitor from the class *i2c_monitor_registers*. This monitor will observe the internal events of the DUT and it will collect the data that is interpreted by it. This means that the monitor be waiting for the DUT to recognize a stop condition internally, and when it does, it will send the data interpreted by the DUT to the scoreboard. The scoreboard will then compare both collected transactions to see if the DUT's behavior matches with the modeled behavior in the testbench.

The scoreboard is represented by a class named *i2c_scoreboard*. This component is responsible for comparing the transactions collected by both monitors and report any mismatches on the behavior of the DUT against the model of the testbench.

The verification of an I2C is similar to the verification planned previously with X-PHY, the testbench will feature monitors watching the serial line and monitors watching over the parallel interface to check the internal registers of the device.

3.4.2 Verifying an I2C master

The verification of an I2C master is very similar to the verification of an I2C slave, it is possible to reuse most of the components except for the driver. The driver has to implement the function of an I2C slave.

The driver will be a class of *i2c_driver_serialslave* and it will go these steps:

1. Request a random transaction from the sequencer, this transaction will contain information about the 8 bit message and about the slave's address
2. Wait for a start condition
3. Collect the address and the rw bit sent to the bus, if it is a match, send an acknowledge
4. Collect or send data depending on the RW bit
5. Wait for a stop condition and repeat the process

The rest of the components can be reused, it is not necessary to create new ones due to the similarities between the I2C slave and I2C master interfaces.

3.4.3 UVM verification components created for the I2C interfaces

To sum up, the table 3.3 represents all the verification components necessary for the verification of an I2C interface (master and slave).

Table 3.3: I2C verification components

Block type	Block class	Parent class
Monitor I2C Serial Slave	i2c_monitor_serialslave	uvm_monitor
Monitor Register	i2c_monitor_registers	uvm_monitor
Scoreboard	i2c_scoreboard	uvm_scoreboard
Driver I2C Serial Master	i2c_driver_serialmaster	uvm_driver
Driver I2C Serial Slave	i2c_driver_serialslave	uvm_driver
Transaction	i2c_trans	uvm_sequence_item
Sequence	i2c_sequence	uvm_sequence
Sequencer	i2c_sequencer	uvm_sequencer
Agent config	i2c_config	uvm_object

The figure 3.18 represents a class tree of the created components.

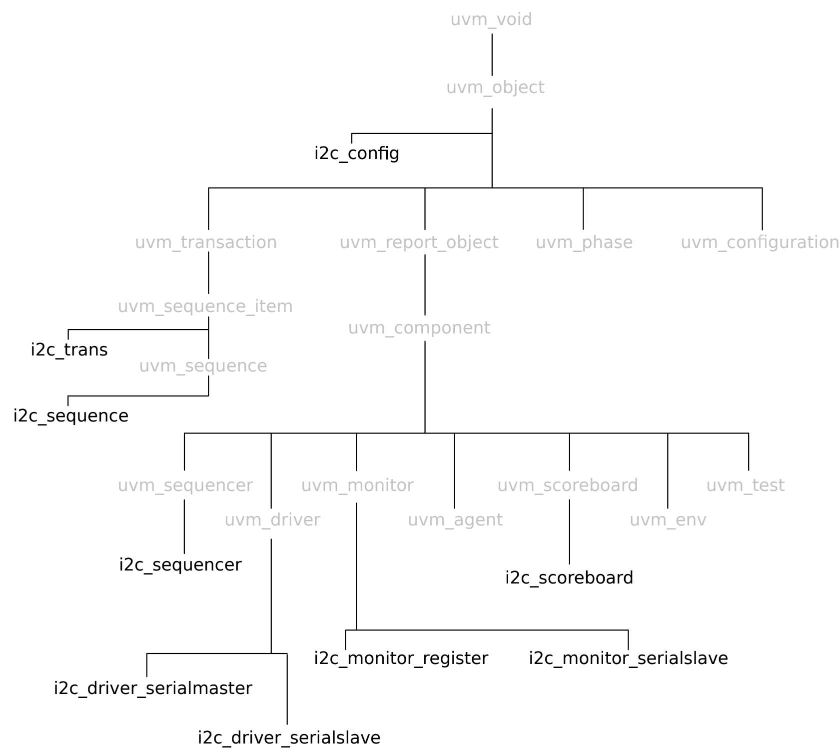

Figure 3.18: Class tree of the created components for a possible I2C testbench

These components will be used to build a verification environment of the device that will be explained in section 3.5.

3.5 SOC Overview

As it was already mentioned, this thesis is being developed within the context of Synopsys' work with communication protocols, so it was not possible to use a real industrial interface as a starting point for the verification environment.

But in order to demonstrate the situations that the testbench has to support, it was created a device denominated of *SOC* as a replacement to the interface.

For this device it's considered a video-surveillance system of a room. The SOC features 3 I2C interfaces: one slave interface and two master interfaces. A fictional ultra-sound sensor and a video camera would be connected to the slave interfaces. A fictional microcontroller would be connected to the slave interface and it would constantly query for video and audio data from the other devices.

The three interfaces will represent the different lanes of the X-PHY. The I2C slave will represent the RX lane while the other two will represent the TX lanes. The two I2C master interfaces will operate at a different speeds: one of them will operate at 400 Kbit/s, representing a low-speed lane, while the other will operate at 1 Mbit/s, representing a high-speed lane.

The device will feature a single clock that can only generate a certain frequency at a given time, this means that both master interfaces will have to wait for one another in order to transmit information.

The SOC is represented on figure 3.19.

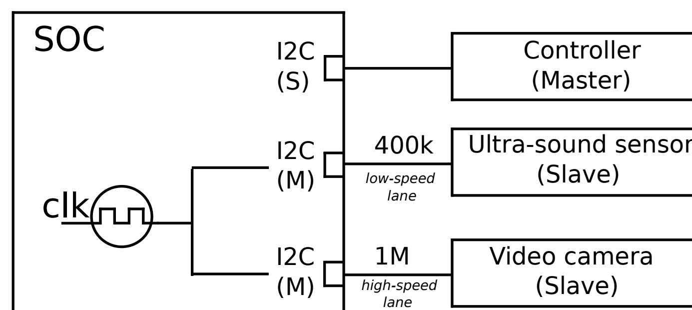

Figure 3.19: Overview of the created SOC

In this scenario, a controller is a device that intends to collect information about the state of the room, so it's constantly querying the SOC for sound and video data. By default, the SOC is collecting data from the ultra-sound sensor (low-speed lane), so when a read request comes from the controller, it's the data from the ultra-sound sensor that it's sent to the controller.

By sending a write command to the bus, the controller can change the lane from the SOC that it's active. The write command with the data 0x01 activates the low-speed lane, while the write command with the data 0x02 activates the high-speed lane.

In the figure 3.20 it's represented a situation where the controller sends the 0x01 command to the SOC, activating the low-speed lane.

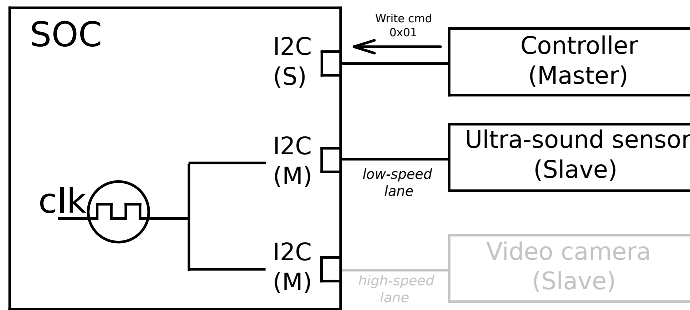

Figure 3.20: Activating the low-speed lane of the SOC

The SOC is now collecting sound samples from the ultra-sound sensor and the controller is aware of the change of states that happened in the DUT, so it can start now to request for sound samples from the sensor.

This situation is represented on figure 3.21.

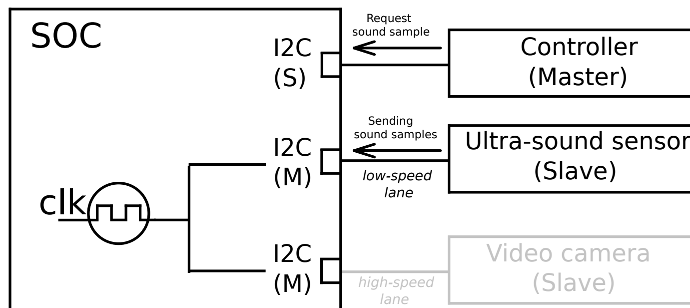

Figure 3.21: Sending sound samples to the controller

After the controller received some sound samples, it can find suspicious activity on the room that is being surveilled. When this happens, it will start to collect video samples from the video camera, so it will request a change of lane to the SOC. This is represented on figure 3.22.

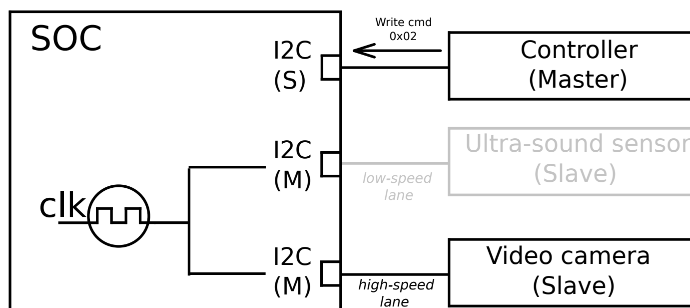

Figure 3.22: Activating the high-speed lane of the SOC

The high-speed lane will be activated and then the controller will start to request for video samples, like in figure 3.23.

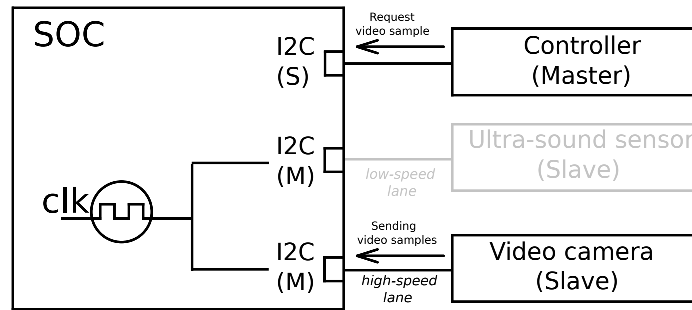

Figure 3.23: Sending video samples to the controller

To sum up, the created SOC will start by collecting data from the low-speed lane. At the same time it will receive instructions through the controller lane. This lane is always active and it can request the SOC to send video samples. Due to the fact that the clock is already being used by the low-speed lane, this lane will have to be deactivated in order to activate the high-speed lane. The SOC will cycle between the low-speed lane and the high-speed lane depending on the requests made by the controller.

This behavior will be used as a model for the verification environment to be developed in this thesis. The section 3.6 will identify potential features to be included in the verification environment and the chapter 4 will fully describe the developed environment.

3.6 SOC Verification Plan

The verification of this device aims for two main objectives: Verification of each I2C interface and verification of the dependency between connected devices. This means that the data transmitted and received by each I2C bus will have to be correctly interpreted by the DUT and that each interface will have to respect the lane constraints defined in chapter 3.5

The components created in section 3.4 will be needed in order to verify the SOC. The table 3.4 presents all the components used for the I2C verification.

Table 3.4: I2C verification components

Block type	Block class
Monitor I2C Serial Slave	i2c_monitor_serialslave
Monitor Register	i2c_monitor_registers
Scoreboard	i2c_scoreboard
Driver I2C Serial Master	i2c_driver_serialmaster
Driver I2C Serial Slave	i2c_driver_serialslave
Transaction	i2c_trans
Sequence	i2c_sequence
Sequencer	i2c_sequencer
Agent config	i2c_config

These components will be used to design an approach to a testbench for the SOC.

3.6.1 Testing the slave interface

For this situation, only one interface will be tested: the I2C Slave. This slave interface is the main communication interface of the SOC, it is from where the whole device is going to be controlled, so it is important to start building a verification system for this interface.

The verification for this situation is very similar to the verification seen on chapter 3.4. Combining the described environment for the X-PHY verification on chapter 3.2 with the components of chapter 3.4.1 it is possible to build a working environment like the one described in figure 3.24.

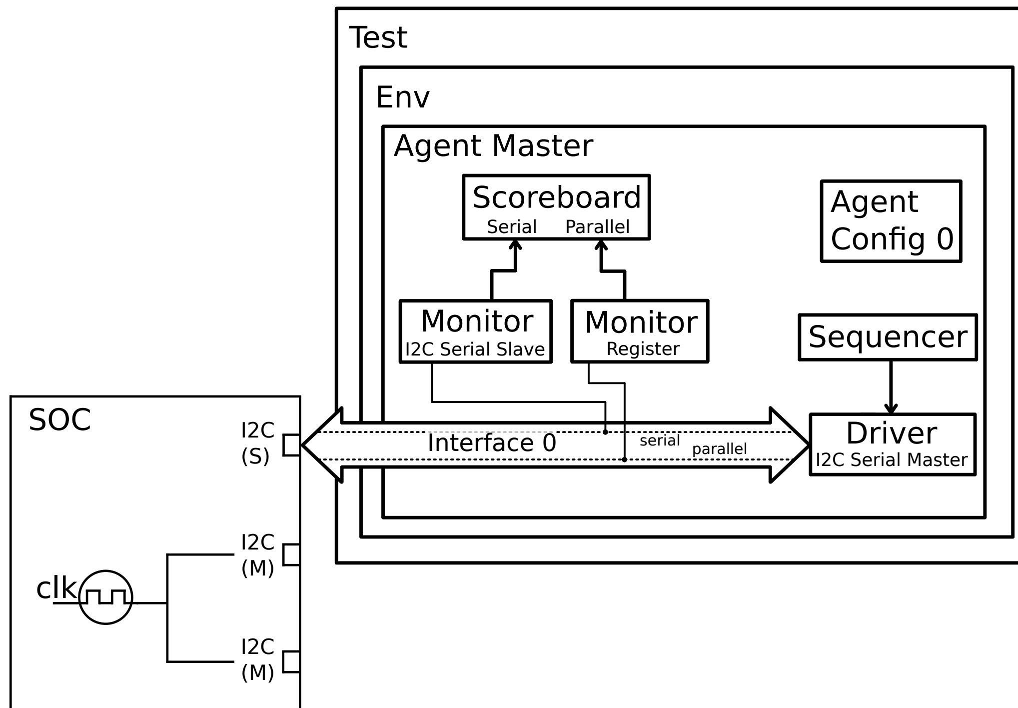

Figure 3.24: Testbench for one I2C interface

Using the monitor, the scoreboard, the agent config block, the sequencer and the I2C master driver, it will be possible to build an agent that emulates an I2C Master device to interact with the slave interface of the SOC.

The rest of the components will work as described in chapter 3.4.

3.6.2 Testing the slave interface and the low-speed lane

Another agent will be added to test one of the I2C-Master interfaces. This slave agent will created by reusing the components created previously, a simple instantiation into a new agent will be enough to replicate some of the intended functionality but for this interface it will be used a slave I2C driver instead of a master one.

The figure 3.25 represents the two agents present in the testbench.

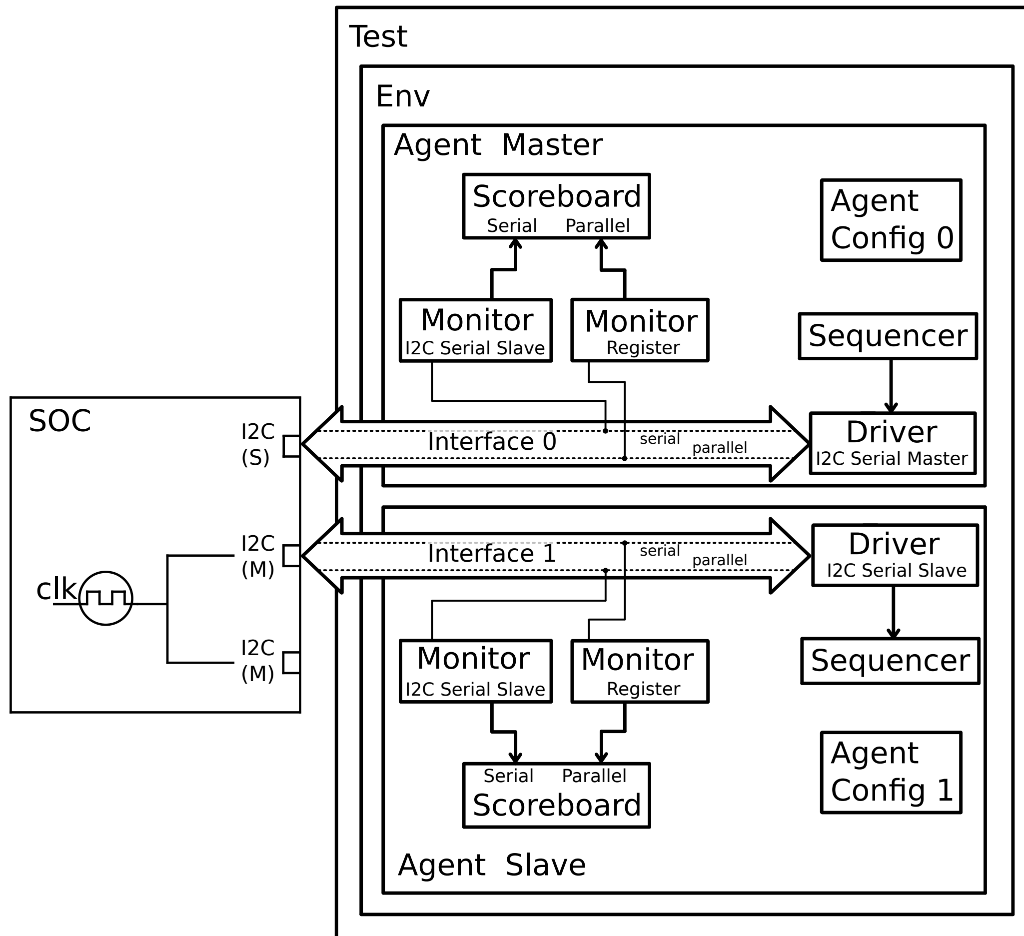

Figure 3.25: Testbench for two I2C interfaces

In this figure, it is possible to see the utility of the bit *active_agent* available in the *Agent Config* blocks. The two agents represent two different tests and the mentioned bit enables, or disables, each agent from the testbench. This means that if it is not desirable to execute one of the tests, it is very easy to disable one agent from the test without rewritting the whole testbench.

With *agent master* it is possible to test the I2C-Slave interface, and the *slave agent* was added in order to test one of the master interfaces. The newest agent is very similar to the first one, except for the driver, which is replaced in order to interact with the master interface.

At the moment, there are two agents testing the DUT at the same time. But these two agents are independent of each other, which means that if, by any chance, the *agent master* generates a transaction that drives the SOC to disable the low-speed lane and enable the high-speed test, the current *agent slave* is rendered useless because the interface is not active.

Due to the nature of the *agent master*, by controlling the DUT configuration through the slave interface, it contains crucial information about the DUT's state of operation. That information could be used to dinamically enable or disable agents. As an example, if the *agent master* writes

the value 0x02 into the bus, the SOC will change its state of operation from low-speed to high-speed. This information should be sent to a block that stays at the same level as the agents and that controls them, an *agent manager*.

The figure 3.26 represents a testbench with an agent manager.

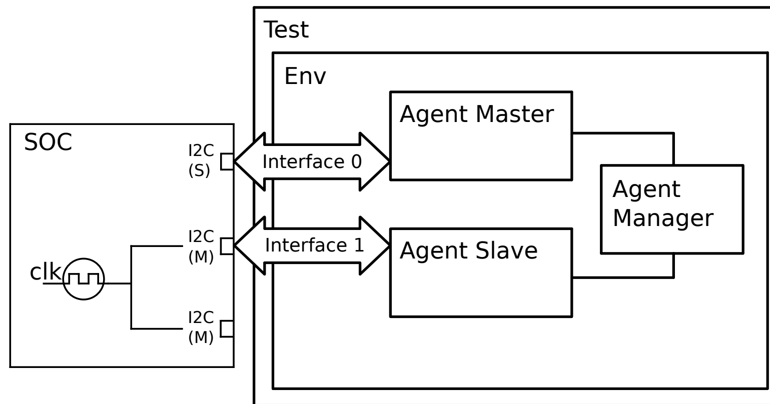

Figure 3.26: Testbench for the SOC with an agent manager

The agent manager should also be able to automatically adapt itself to any number of agents that it is necessay to add to the testbench.

3.6.3 Testing the slave interface and the low-speed and high-speed lanes

If it is desirable to test the remaining master interface as well, it's only necessary to create a new instance of the same slave agent used previously. The figure 5.3 represents the complete testbench for the SOC.

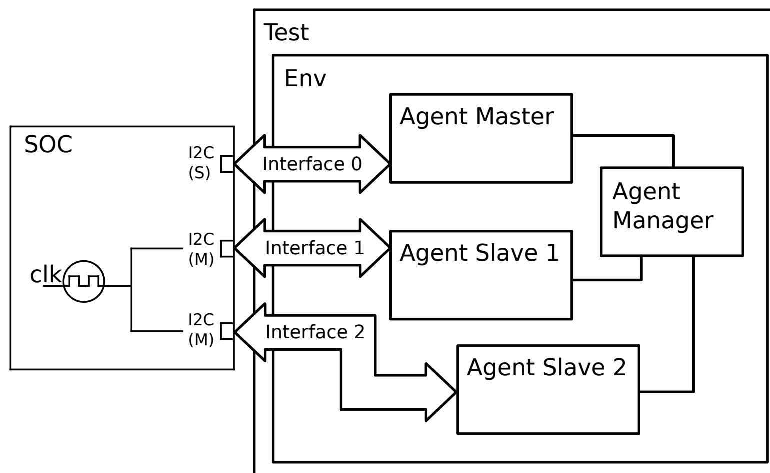

Figure 3.27: Complete testbench for the SOC

In this situation, the agent manager enables and disables the slave agents automatically. The default operation of the SOC is to collect sound samples from the low-speed lane, so if the *agent*

master generates request, the SOC makes a normal response. The figure 3.28 represents this situation.

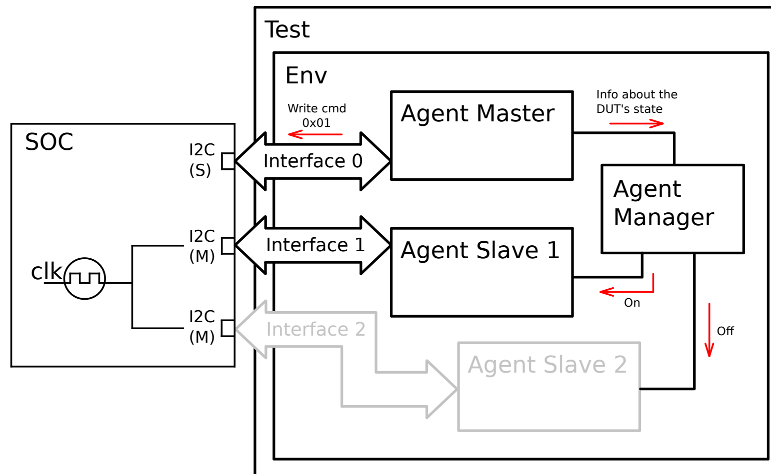

Figure 3.28: Testbench for the SOC with *Agent Slave 2* disabled

If a write cmd 0x01 is generated, the *agent master* will inform the agent manager which will disable the *agent slave 2*.

However, if the *agent master* generates a request to activate the high-speed lane (this is, if a write cmd 0x02 is generated), it will also inform the agent manager about this situation, which in its turn will enable the *agent slave #2* and disable the *agent slave #1* (figure 3.29).

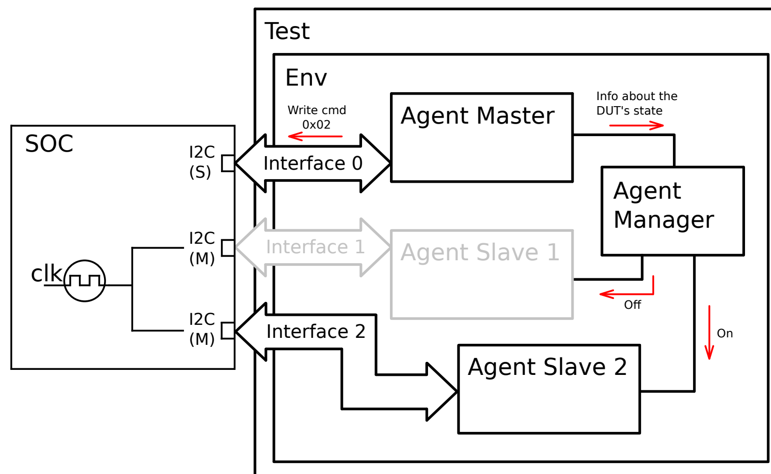

Figure 3.29: Testbench for the SOC with *Agent Slave 1* disabled

This is the expected behavior of the SOC as it was mentioned in 3.5. The agent manager must have a state machine that matches the behavior of the DUT. It should be noted that the agent manager does not have to disable agents, this was just an example. Each agent could have programmed any other behavior, like waiting states or burst transmissions. It depends on the engineer to decide what kind of behavior that the testbench should have.

3.7 Conclusion

The current chapter gave an overview about some of the features that high-speed communication protocols have and the kind of situations that a testbench should support in order to fully verify the functionalities of those protocols.

A device denominated SOC was created to demonstrate the underlying architecture of the devices that implement such high-speed protocols. This device also served to a model an approach to a possible testbench capable of supporting this category of devices.

Some features that the testbench should support include:

- Each agent representing a test
- The ability to add or remove agents manually
- To let the testbench enable or disable agents automatically depending on the test being executed to the DUT
- Each agent supports multiple states to be defined by the agent manager. As an example, in the SOC, each agent only had two states: enable and disabled
- Support multiple DUT configurations. As an example, if the SOC had a third master I2C interface, it would be easy to reuse the components that we already have in the testbench and extend the agent manager to one more agent
- To force directed tests

This list will serve as points to be kept in mind while developing the testbench. The chapter 4 will provide an overview and documentation of the developed testbench.

Chapter 4

The Verification Environment

The previous chapter provided some of the situations that an ideal verification environment has to handle in order to fully verify a device under test. By taking into account the analysis made to communication protocols, it was created a new verification environment built on top of SystemVerilog and UVM. This chapter will present the necessary documentation and the design decisions made during the conception of the environment: the components, the relationship between them, how they communicate and the work flow of the environment in order to adapt it to new devices.

The chapter will start by giving an overview of the testbench: the purpose of the testbench, the components and the expected behavior. Then it will move to the description of each component: a deep explanation of each component will be given, which will serve as a manual of the created classes. And finally, it will end with the necessary steps to use the verification in other situations. Some of the explanations are also included in the comments of the source code of the testbench.

The verification environment presented here is be a composition of classes and verification guidelines studied under the scope of Synopsys' work with communication protocols, so this might not represent a true generic and universal approach to UVM testbenches.

4.1 Testbench Overview

As it was mentioned previously, the most basic unit of the testbench is the agent. An agent is be composed of drivers, sequencers, monitors, scoreboards and configuration blocks and they all represent a test being exercised to the DUT. This means that two different agents represent two different tests.

It is possible to manually enabled and disabled each agent by modifiying the respective configuration block. As a result, the tests done to the DUT can be easily changed without having to rewrite the whole environment.

But an interesting feature is to let the environment to dinamically enable and disable each agent during the execution of the test, without the intervention of the engineer. It would increase the coverage of the test without having to worry with which test was being executed. This kind of automatization is possible due to the introduction of an agent manager.

All these components establish the most basic structure of the intended verification environment. A top level view of the environment can be seen in figure 4.1.

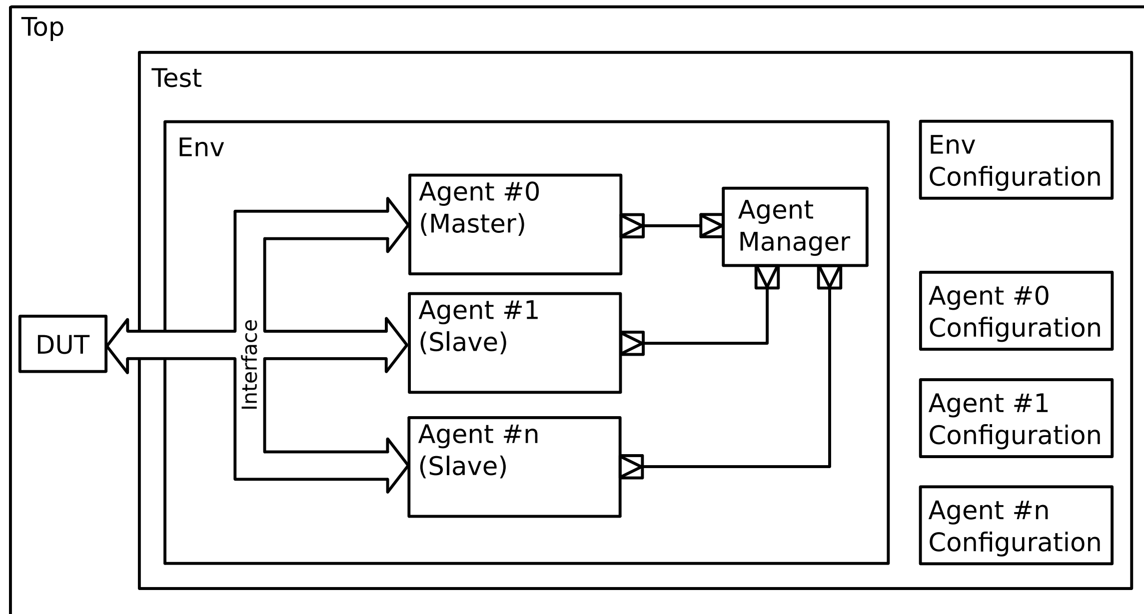

Figure 4.1: A top level view of the verification environment

The agent manager is responsible for collecting information of certain agents in order to control others, so it is necessary to establish a master-slave relationship between the agents. A master agent is an agent responsible for testing the core functionality of the device, like the interface with registers. This agent sends information about the DUT's state to the agent manager and the agent manager enables and disable agents depending on that information.

As an example, it is considered an audio codec with 2 analog inputs (*input #1* and *input #2*) and 2 analog outputs (*output #1* and *output #2*). Besides the analog interface, it is also assumed that there is a digital serial interface that with access to the codec's registers. In this scenario, there would be 3 agents: a master agent that would be attached to the digital serial interface and two slave agents that would attach to one analog input and one analog output.

The master agent would check if the serial interface would be operating correctly. This test would mean writing consecutively into the registers and, as a result, changing the behavior of the analog inputs and outputs. The slave agents would be attached to their own analog input and output and they would generate an analog wave and verify the result in the output of the DUT.

If, by any chance, the master agent would generate a sequence that would program the DUT's register to disable the *input #1* and the *output #1*, one of the agents would be rendered useless. For that reason, the master agent would inform the agent manager about the situation and the agent manager would disable the respective agent.

On the other hand, if the master agent would generate a sequence that would activate the *input #2* and the *output #2*, the other slave agent could be enabled in order to test the analog inputs/outputs of the DUT.

This situation is represented on figure 4.2.

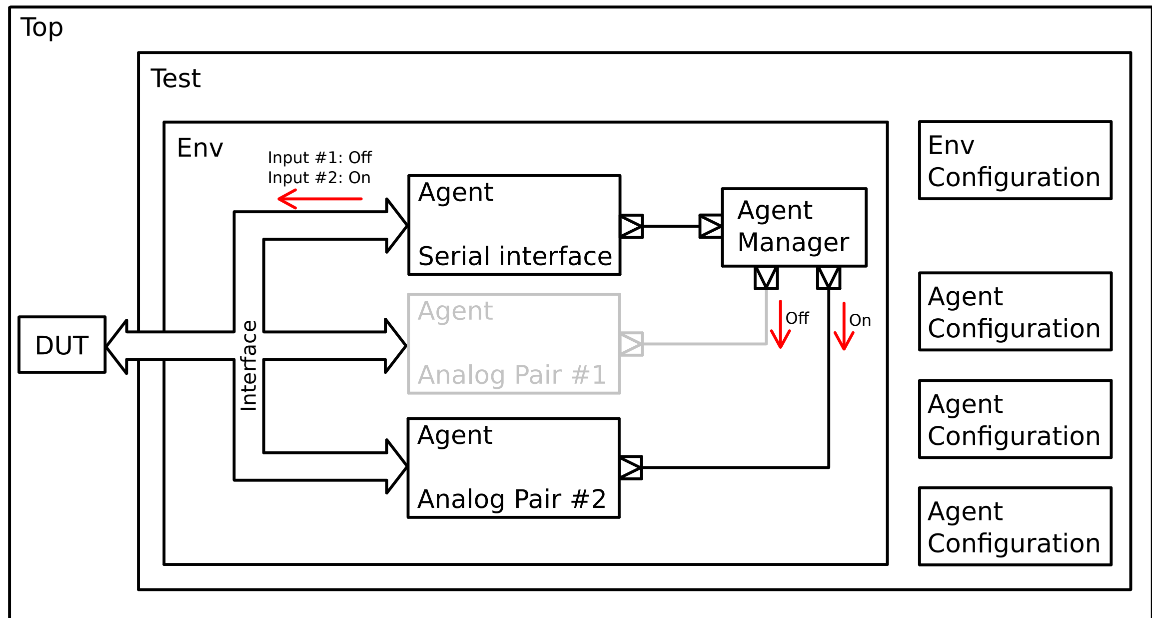

Figure 4.2: A top level view of the verification environment

Each agent is configured with the settings defined in *Agent Configuration*. This block holds the virtual interface to be connected to the agent, it holds the settings to force random or direct tests and it also designates if the agent is active or not during compilation time.

The environment is configured by the block *Env Configuration*, this block holds the objects for the configurations of each agents and the number of each master and slave agents. The number of agents is used to configure the *agent manager*.

The configuration of the environment and of the agents is first declared in the *Test* class. This way, the testbench can be configured in a single file, which increases simplicity in its usage.

The communication between the agents and the agent manager is done using TLM-2.0 sockets. The agent manager receives information from the master agents by a target socket and sends information to the slave agents using an initiator socket. These sockets were grouped in two different classes to ease their utilization: *socket_slave_container* and *socket_master_container*, respectively.

As it was mentioned previously, the agents are divided in two categories: master agents and slave agents. Master agents are characterized by having at least one component that sends information about the DUT's state to the agent manager, this kind of component is denominated of *master component*. On the other hand, slave agents are agents whose components are affected by the information sent to the agent manager, these components are denominated of *slave components*.

The figure 4.14 represents the components of a slave agent.

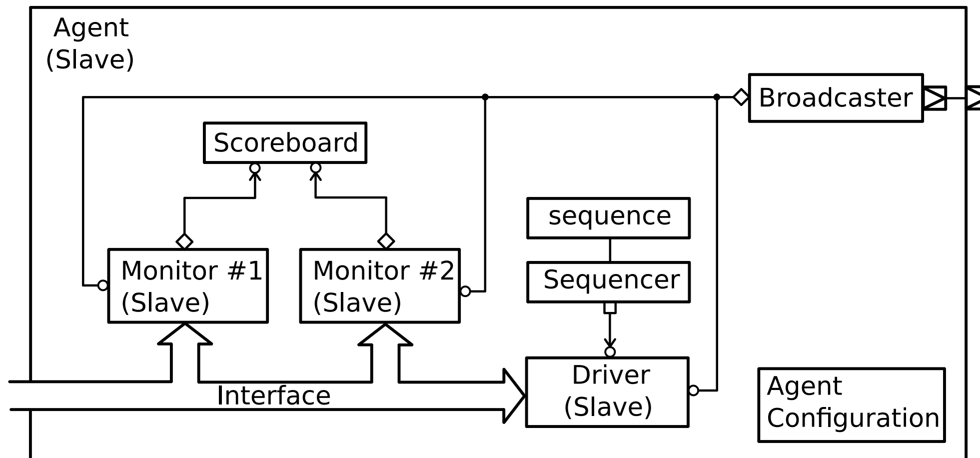

Figure 4.3: A top level view of a slave agent

The agent's socket is connected to a block named *broadcaster*. The purpose of the broadcaster is to take messages from the agent manager and send them to monitors and drivers, so they can be aware of the agent's state.

A normal socket only allows for a maximum of one connected object at the same time, so it is unfit to send information to multiple objects at the same time. However, analysis ports are fit for this behavior: each time an export connects to an analysis port, the analysis ports adds it to an internal list. When a message is sent through the analysis port, the port cycles through the list and relays the message to each connected export. [3, p. 16]

So the broadcaster was created in order to convert a socket connection into an analysis port, allowing to send messages from the agent manager to multiple components.

Unlike a slave agent, the structure of the master agent is more simple. In this agent, there is not a broadcaster block. It is mostly due to the fact that the socket connection of the master agent connects to only one component that is aware of the change of states that happens in the DUT, usually the master monitor. This master monitor is the component that feeds information about the DUT's configuration to the agent manager, it is usually a monitor that watches the serial line for the most important communication.

The figure 4.13 represents the structure of a master agent.

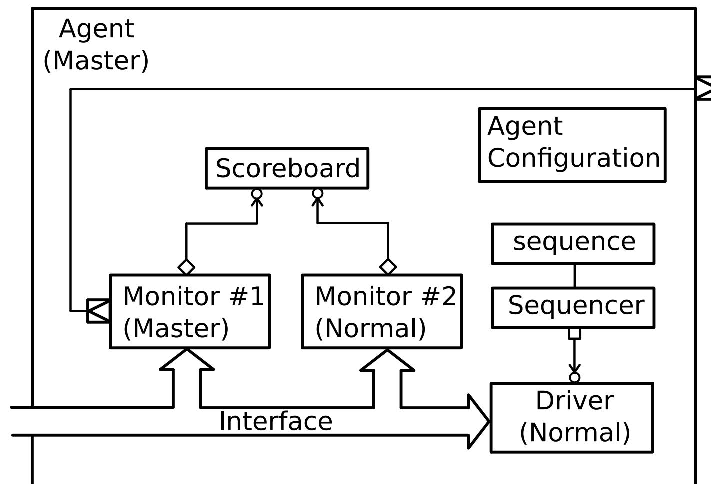

Figure 4.4: A top level view of a master agent

The sockets present in both agents, master and slave, are denominated of *passthrough sockets*. They represent sockets that relay information to other socket, they are usefull in a way that help to isolate sub-components within the components while keeping the connections intact.

This testbench provides a basic architecture that can be easily extended and modified to a DUT's needs. Each of the mentioned blocks are represented by SystemVerilog classes that allow for this kind of modification. A top level view of the testbench with the classes' names can be seen on figure 4.5.

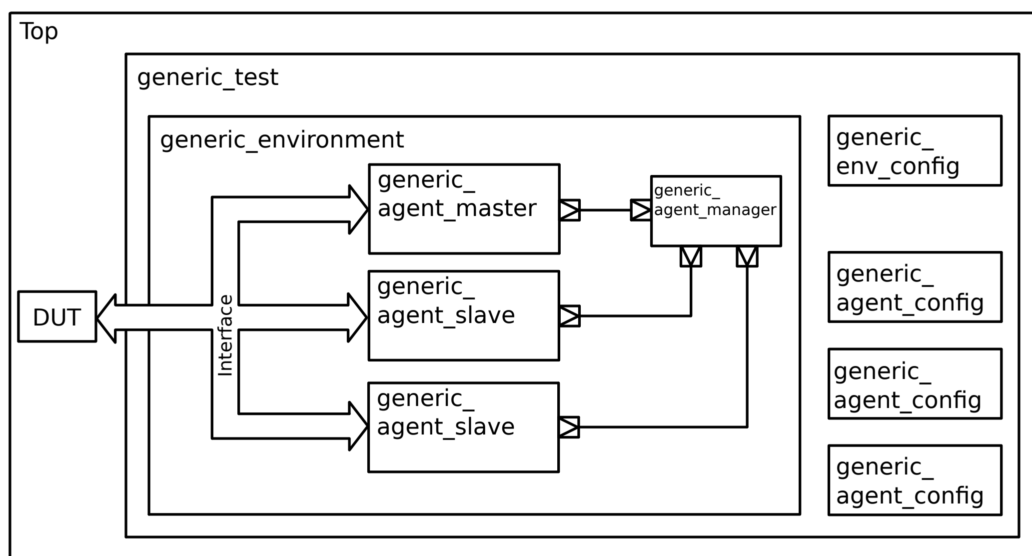

Figure 4.5: A top level view of the verification environment with class names

The current section served to present an overview of the functionality of the testbench: what is supposed to do, how classes are organized in the environment and how they communicate with

each other.

Starting from the next section, it will be given a deeper explanation of the components of the testbench and how to use them:

- The section 4.1.1 will summarize every component in the testbench, their parent classes and their path within the folder hierarchy.
- The section 4.1.2 will present an overview of the file organization of the testbench.
- The explanation of each component will start in the section 4.2. These sections will describe the structure of each block as it is structured in the source code.
- The section 4.13 will define a workflow list that should provide a reference point to adapt this testbench to other devices.

By the end of section 4, a basic understanding about the verification environment should already be given and an example of its application will be demonstrated in chapters 5 and 6.

4.1.1 Class table

The table 4.1 represents a list of all the classes created for the testbench.

Block type	Block class	Parent class	File	Chapter
Configuration	generic_agent_config	*uvm_object*	*config/generic_agent_config.sv*	4.2.1
Configuration	generic_env_config	*uvm_object*	*config/generic_env_config.sv*	4.2.2
Test	generic_test	*uvm_test*	*test/generic_test.sv*	4.3
Environment	generic_environment	*uvm_environment*	*env/generic_environment.sv*	4.4
Agent Manager	generic_agent_manager	*uvm_component*	*general_class/generic_agent_manager.sv*	4.5
Info Block	generic_info_block	*uvm_object*	*general_class/generic_info_block.tex*	4.6
Socket container	socket_master_container	*uvm_component*	*general_class/sockets.sv*	4.7
Socket container	socket_slave_container	*uvm_component*	*general_class/sockets.sv*	4.7
Agent	generic_agent_master	*uvm_agent*	*env/generic_agent_master.sv*	4.8.1
Agent	generic_agent_slave	*uvm_agent*	*env/generic_agent_slave.sv*	4.8.2
Broadcaster	broadcaster	*uvm_component*	*general_class/broadcaster.sv*	4.9
Monitor	generic_monitor_master	*uvm_monitor*	*monitors/generic_monitor_master.sv*	4.10.1
Monitor	generic_monitor_slave	*uvm_monitor*	*monitors/generic_monitor_slave.sv*	4.10.2
Driver	generic_driver_slave	*uvm_driver*	*drivers/generic_driver_slave.sv*	4.11

Table 4.1: Elements of the class *generic_agent_config*

The code of the classes is partitioned with comments to make clear at the first sight how each class is composed. This composition will be explained the the format of tables in the following sections.

The class tree of the verification environment is represented on figure 4.6.

Figure 4.6: Class tree of the created testbench

4.1.2 File system

The organization of the verification environment on the file system is shown on the file tree below.

```
generic_tb/
├── agents/
│   ├── generic_agent_master.sv
│   └── generic_agent_slave.sv
├── config/
│   ├── generic_agent_config.sv
│   └── generic_env_config.sv
├── drivers/
│   └── generic_driver_slave.sv
├── dut/
├── envs/
│   └── generic_env.sv
├── general_class/
│   ├── broadcaster.sv
│   ├── generic_agent_manager.sv
│   └── socket.sv
├── interfaces/
├── monitors/
│   ├── generic_monitor_master.sv
│   └── generic_monitor_slave.sv
├── pkg/
│   └── generic_pkg.sv
├── scoreboards/
│   └── generic_scoreboard.sv
├── sequences/
├── tests/
├── Makefile.vcs
└── top_module.sv
```

All the source files from the DUT must be placed in the *generic_tb/dut/* directory and all the virtual interfaces must go into the *generic_tb/interfaces/* directory.

The new components added to the testbench must be place into their own directory (agents in the *generic_tb/agents/* , monitors in the *generic_tb/monitors/* and so on) and all the new files must be added to the *generic_tb/pkg/generic_pkg.sv*. This file contains a list of all the files from the testbench to be included during the compilation.

The file *generic_tb/top_module.sv* is the file that connects the DUT and the testbench and the execution of the testbench is done using the provided Makefile.

4.2 Configuration Blocks

There are two main base configuration blocks:

- The agent configuration block: generic_agent_config
- The env configuration block: generic_env_config

They are highlighted in figure 4.7.

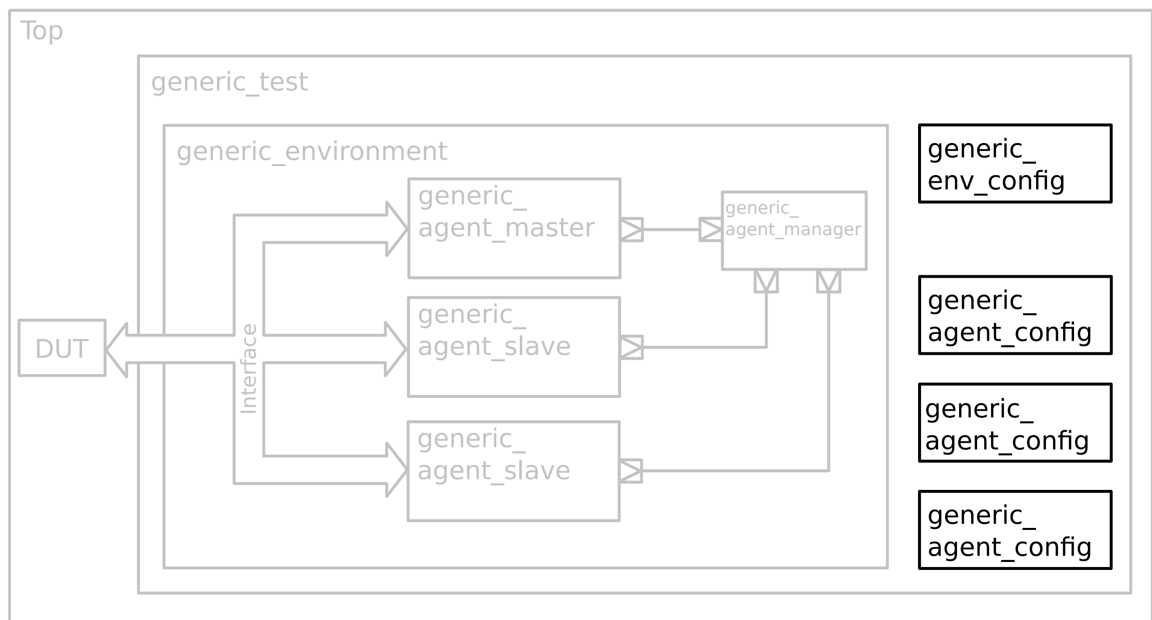

Figure 4.7: Configuration blocks of the verification environment

Both of these blocks are extended from the default class *uvm_object* and they are required in order to configure the environment and the agents within the environment. They can be found in the directory *src/config/*.

These blocks are created in the *test* class and then, they are saved in the UVM configuration database (*uvm_config_db*) to be loaded later by each object.

4.2.1 Agent configuration block

The class *generic_agent_config* holds the necessary configuration of each agent in order to to test the DUT. This configuration includes three main aspects: variables that specify the state of the agent, the virtual interface for the comunication with the device and any variables needed to force a direct test.

These configurations affect mostly the monitors, the drivers and the sequencers. The monitors and the drivers will get the virtual interface through this block and the sequencers will get information about the test (like the randomness, number of transactions to be generated and variables override).

The class constructor and the necessary UVM macros are also present in this class.

This configuration is unique to every agent of the testbench, so for each different agent instantiated within the environment, a new object of this class must be created. After the configuration is defined in the *test* class, it is saved in *uvm_config_db* and then loaded inside each agent. More information can be consulted in the chapters 4.3 and 4.8.

A sum up of this class can be seen in the table 4.2.

generic_agent_config	
Field	Description
Default variables	bit *agent_active* int *random* int *num_trans*
Virtual interfaces	Interfaces for the communication with the DUT
Variables for direct testing	Any variables needed to force direct tests
Constructor	Typical class constructor
UVM Macros	Typical UVM macros

Table 4.2: Elements of the class *generic_agent_config*

The '*Default Variables*' are variables already available that specify the behavior of the corresponding agent. A brief description of each will follow:

- bit *agent_active* - This variable tells to the environment to build all the components inside of the agent and it connects the agent to the *agent_manager*. If the variable is in disable state, the agent won't be connected to the *agent_manager* and the components inside of this agent will be ignored and they won't be created. Possible states:
  - 1'b0 - Agent disabled
  - 1'b1 - Agent enabled
- int *random* - This variable states wether the agent will do a random test or not, and if it will do one, it can also state the randomness of the test. The variable is not a bit type but an integer type. Unlike the variable *agent_active*, this one doesn't force anything in the testbench, it is up to the verification engineer to use this variable as he sees fit, it is more a guideline than anything else. Ideally, the value '0' means no randomness and then variables for direct testing must be set but the values bigger than '0' would specify the type of randomness. An example will follow:
  - 0 - No randomness. Aditional variables specific to the DUT must be set as well as *num_trans*
  - 1 - Some randomness. The sequencer will generate a number of transactions defined by *num_trans* and limited with a set of constraints
  - 1 - More randomness. The sequencer will generate a number os transactions defined by *num_trans* but they won't be limited by constraints

  - 2 - Total randomness. The sequencer will generate a random number of transactions (it won't be affected by *num_trans*) and they won't be limited by constraints

- int *num_trans* - This variable defines the number of transactions to be generated in the sequencers of the agents. For more information, check the description of the '*random*' variable above.

The '*Virtual interfaces*' is a field reserved for all the interfaces that are used by the agent. They are defined in the classes derived from '*generic_agent_config*'.

The '*Variables for direct testing*' field is reserved for cases when it is necessary to force a certain test to the DUT. For example, if we are testing an I2C interface, a random test would randomize the slave addresses that are sent through the communication bus but this gives the option to force the address to a specific value.

Both '*Virtual interfaces*' and '*Variables for direct testing*' fields aren't implemented by default, they must be filled out in the derived classes from '*generic_agent_config*'. Although the variable from the field '*Default Variables*' are initialized, they must be defined in the derived class too.

4.2.2 Env configuration block

The class *generic_env_config* contains information about the number of master and slaves agents present in the testbench, this information is used by the *agent_manager* to create all the necessary sockets. The class also contains the objects for the configuration of each agent.

The structure of the block can be consulted in the table 4.3.

generic_env_config	
Field	Description
Variables	*n_agents_master* *n_agents_slave*
Agent configurations	Objects for the configuration of each agent
Constructor	Typical class constructor
UVM Macros	Typical UVM macros

Table 4.3: Elements of the class *generic_env_config*

The *Variables* field only contain two variables: *n_agents_master*, the number of master agents, and *n_agents_slave*, the number of slave agents. These variables should indicate the number of agents present in the code, not the number of enabled agents that will be defined in the *test* class. Both these two variables must be defined in the derived classes.

The *Agent configurations* field contains the objects for the configuration block of each agent, these objects are put in the env configuration so that they can be loaded into each agent during the build phase of the *generic_agent*. This field must be filled out in the derived class.

4.3 The Test Block

The test block is derived from the class *uvm_test* and this is where the whole test should be configured. It is highlighted in figure 4.8.

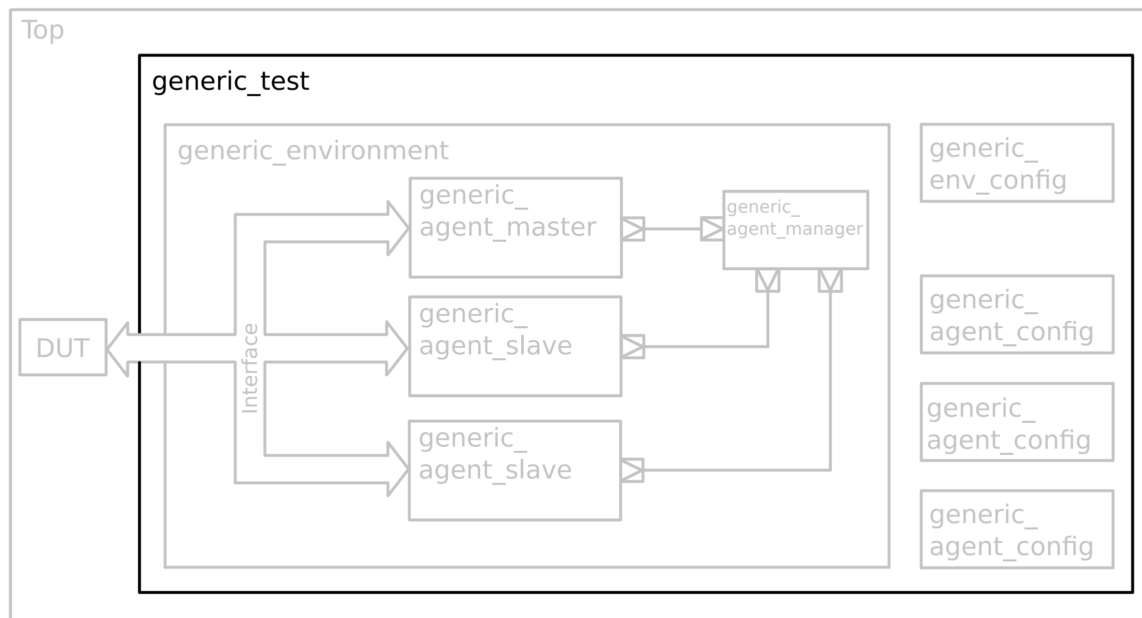

Figure 4.8: Test block of the verification environment

The environment and the configuration blocks for the agent and for the environment are defined in this class, including the type of test itself (if it is random or not, how many transactions will the sequencers generate and what variables will be forced for direct testing). The virtual interfaces for the agents are retrived from the configuration database and submitted to the agent configuration blocks (objects derived from the class *generic_agent_config*).

Along with the source code of the environment, and in the examples below, it is provided a class named *generic_test* but it is just a base example of a typical *test* class, it isn't necessary for new tests to be derived from this class.

A typical *test* class is represented in the table 4.4.

generic_test	
Field	Description
Environment and configuration blocks	Instantiation of the environment and of the configuration blocks
Constructor	Typical class constructor
Build Phase	Creation of the instantiated blocks and configuration of the test
UVM Macros	Typical UVM macros

Table 4.4: Elements of the class *generic_test*

The field "*Environment and configuration blocks*" instantiates the necessary configuration blocks for the environment and for each agent. If there are 3 agents in the testbench, 3 agent configuration objects will be created.

The field "*Build Phase*" is where the whole test is configured by setting up the variables defined in each configuration. This phase also retrieves the virtual interface from the configuration database and saves it into each agent configuration. Each agent configuration is saved in the env configuration block. The file *generic_tb/tests/generic_test.sv* provides some examples for a typical test block.

4.4 The *Env* Block

The class *generic_envinroment* is responsible for configuring the agents of the testbench as well the agent manager. Figure 4.9 represents the environment class.

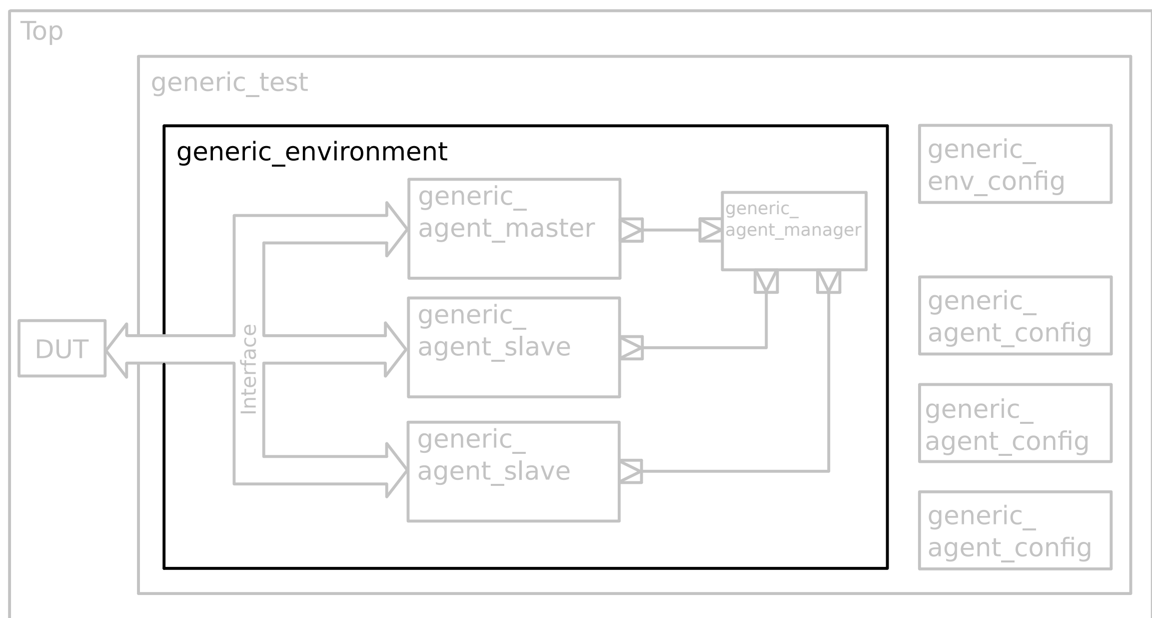

Figure 4.9: *Env* block of the verification environment

The class retrives the *env* configuration block from the database and it checks in every agent configuration block, whether the respective agent is enabled or not. If the agent is enabled, then it is created and its configuration is submitted to the configuration database, (*uvm_config_db*), so that it can be used by each agent in order to configure the monitors, the drivers and sequencers.

The *env* block creates the agent manager and sets up the socket connections from this block to each agent. It also provides a number of dummy socket connections equal to the number of slave agents.

These dummy sockets replace the connection of each disabled agent to the agent manager. They are needed because the agent manager uses the variable *n_agents_slave* available in the *env*

config block, and it won't take into account the number of agents disabled manually in the *test* class.

The testbench assumes that the agents in the code are always be enabled but, at the same time, it provides the ability to override the creation of an agent without removing it from the code, in case of the agent having missing features or unexpected bugs. If the agent is manually disabled from the testbench, the agent manager's socket will attempt to connect to the socket nevertheless but UVM socket initiators require to be connect to one terminator, otherwise the testbench will not execute. So these dummy sockets were arranged for these special cases.

The basic idea is to configure the environment and all the agents in the *test* class and save the agent configurations in the *environment* block. The *environment* will load those configurations and it will build the agents accordingly.

The structure of the *generic_env_class* is represented on table 4.5

generic_env	
Field	Description
Env config	Configuration block for the *env* (class derived from *generic_env_config*)
Agents of the testbench and agent manager	Instatiation of all the agents and of the agent manager
Dummy sockets	Dummy sockets to replace missing socket connections
Constructor	Typical class constructor
Build Phase	Retrieval of the *env* configuration and creation of the agents and of the agent manager
Connect Phase	Connection of the agents to the agent manager
UVM Macros	Typical UVM macros

Table 4.5: Elements of the class *generic_env*

The *Env config* field is destined to the instatiation of the object that holds the *env* configuration, which is an object derived from the class *generic_env_config*. This configuration is then retrieved from the configuration database during the build phase.

The *Agents of the testbench and agent manager* field is reserved to the instantiation of all the agents available in the testbench, as well the agent manager.

The *Dummy Sockets* are the sockets used to replace the connection of missing agents to the agent manager as described in the previous text.

The *Build Phase* retrieves the configuration of the environment and it creates all the agents and the agent manager instantiated in the *Agents of the testbench and agent manager* field. In this phase, it is recommended to check if the agent is available or not by checking the variable *generic_agent_config.agent_active* before building the object.

In the *Connect Phase*, the connections between the agents and the agent manager are made. Like in the *Build Phase*, it is recomended to check if the agent is active before establishing the

connection. If the agent is disable, the connection should be made to a dummy socket instead.

Further examples are demonstrated in the comments of the file *generic_tb/envs/generic_env.sv*.

4.5 The Agent Manager

The *agent manager* block is derived from the class *uvm_component*. It is not derived from the other most known classes like *uvm_sequencer*, *uvm_monitor* or *uvm_driver* because it is neither a sequencer, a monitor or a driver. It is not derived from *uvm_object* either because this class doesn't support some of the necessary features like *phases* and TLM ports and sockets. So it is derived from *uvm_component*, which is the root class for all other UVM components, which means support for *phases* and TLM, and inherits the features from *uvm_object*.

The class is represented in figure 4.10.

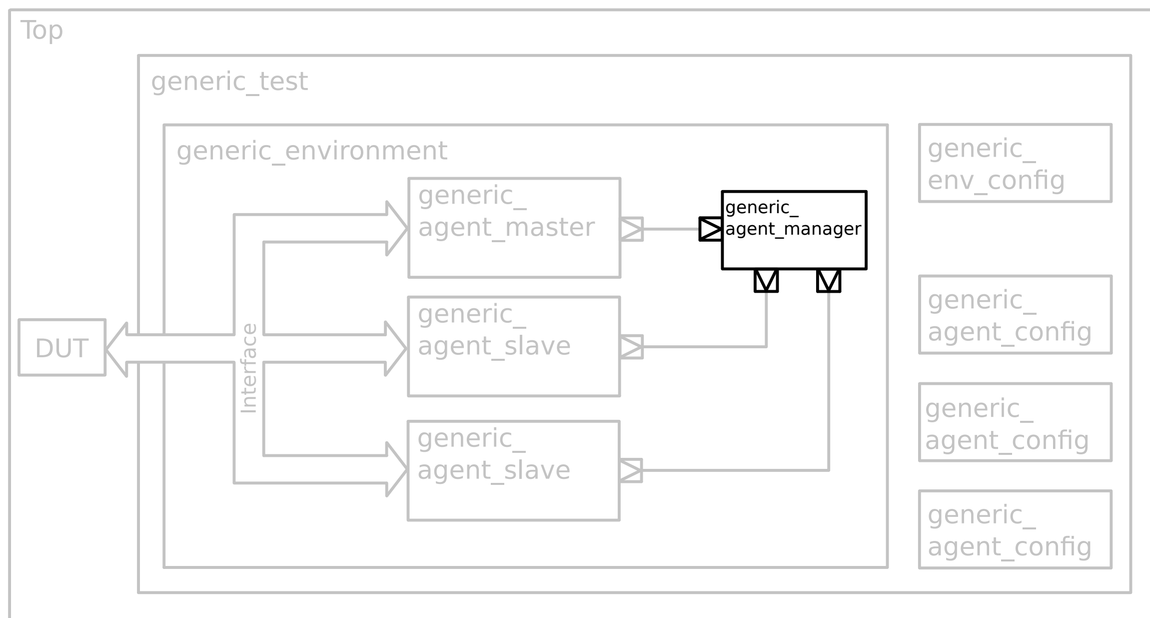

Figure 4.10: Agent manager block of the verification environment

This class receives an object from a master agent, processes it in a state machine and controls the other slave agents with this information. The *generic_agent_manager* can be consulted on the table 4.6.

generic_agent_manager	
Field	Description
Sockets	Instantiation of the array of sockets to be connected to the agents of the environment
Variables	Variables that specify the number of master and slave agents used to create the number of sockets necessary for all the agents
Constructor	Typical class constructor
Build Phase	Creation of the sockets necessary for the testbench
Run Phase	Execution of the state machine
UVM Macros	Typical UVM macros

Table 4.6: Elements of the class *generic_agent_manager*

The *Sockets* field declares two kind of sockets. A *slave socket* and a *master socket*. The slave sockets receive information from the master agents and the master sockets send information to the slave agents. These sockets are objects from the classes *socket_slave_container* and *socket_master_container* respectively. They will be mentioned in section 4.7.

The *Variables* field declares two variables: *n_agents_master* and *n_agents_slave*. The values of these two variables are specified by the class *generic_env* and they indicate the number of master agents and slave agents in the testbench. They are used to automatically create the number of sockets for all the agents during the build phase. In this field should also be declared the object that is going to be used by the state machine of this agent manager in order to control the slave agents.

In the *build phase*, it is used the variables *n_agents_slave* and *n_agents_master*, which are defined by the *generic_env* class, to create all the sockets. So if there are 3 slave agents in the testbench, it will be created 3 master sockets to send them the information about their state.

In the *run phase*, it is executed the state machine of the agent manager. It is represented by a function *state_machine()* which should be overridden in the derived classes from *generic_agent_manager*.

The derived classes from this one should add the object that contains information necessary to the state machine in order to control the agents and the state machine itself.

An example of the class from the current section can be found in the comments of the file *generic_tb/general_class/generic_agent_manager.sv*.

4.6 Generic Info Block

The class *generic_info_block* is derived from *uvm_object* and it represents the parent class of the object that is going to be sent to the agent manager in order to control the slave agents. The

classes derived from *generic_info_block* should contain information about the changes to the DUT, usually, the registers that are going to be modified.

The structure of the block if very simple, it can be consulted in the table 4.7.

generic_info_block	
Field	Description
Variables	Variables that model the changes made to the DUT
Constructor	Typical class constructor
UVM Macros	Typical UVM macros

Table 4.7: Elements of the class *generic_info_block*

An example of variables to be filled in the "*Variables*" field are the register variable and the data to be written into that register. The state machine of the agent manager could be programmed to react to this information in a way that disables and enables the necessary agents for the test.

4.7 Socket containers

The socket containers are two classes that have initiator sockets (denominated here as socket masters) and target socket (denominated here as socket slaves), these classes are *socket_master _container* and *socket_slave_container* respectively.

Just like the component *generic_agent_manager* (chapter 4.5), these classes are derived from *uvm_component*.

By having these sockets in classes, it becomes easier to implement them in a new component, this way it is enough to call the class socket container and the socket is ready to be used.

The figure 4.11 represents the socket containers on the agent manager.

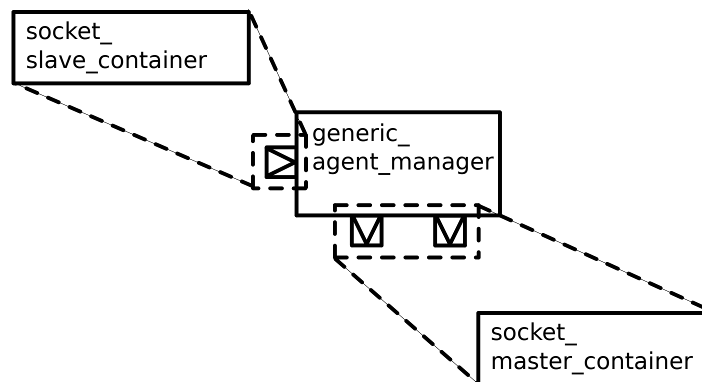

Figure 4.11: Sockets from the agent manager

Firstly, it will follow an explanation about the socket container that is used to receive messages from the master agent, and then it will be explained the socket container used to send messages to the slave agents.

The class *socket_slave_container* is represented in the table 4.8;

socket_slave_container	
Field	Description
Target socket	Instantiation of the target socket that passes around an object derived from *generic_info_block*
Variables	Instatiation of some of the objects needed for the socket
Constructor	Typical class constructor
Task *b_transport()*	Implementation of the task to be executed when the socket is used
UVM Macros	Typical UVM macros

Table 4.8: Elements of the class *socket_slave_container*

The *socket_slave_container* is responsible for getting messages from the master agents as it implements a target socket. It contains a variable named *notification* that changes its value from 1'b0 to 1'b1 every time the function *b_transport()* is summoned by a initiator socket. The socket receives a class named *generic_info* and the function *b_transport()* copies it to a local object. To access to the information from this object, the component that creates this container, must cast the object *generic_info* to the desired variable type.

An example will follow in the code 4.1.

```
1   class i2c_agent_manager extends generic_agent_manager;
2           socket_slave_container  socket_slave    ;
3           generic_info            i2c_info        ;
4
5           //Code for the constructor and other functions
6
7           virtual task run_phase(uvm_phase phase);
8                   @(socket_slave.notification);
9                   $cast(i2c_info, socket_slave.generic_info);
10
11                  //Code for i2c_info usage goes here
12          endtask: run_phase
13  endclass: i2c_agent_manager
```

Code 4.1: Code for the usage of socket slave container

This keeps this container generic enough for any testbench.

The run phase waits for a notification of the container to trigger, meaning that there is data available to be read, then the task casts the variable *generic_info* to the variable *i2c_info* in order to access the fields of the original information.

The *Target socket* field instantiates the target socket. This socket passes around an object derived from *generic_info_block*, so in order to use these containers, we have to derive a class from that class.

The *Variables* field contains the instantiation of the object to be received through the socket, a *delay* object that is necessary for the socket as well and a variable *notification*. This variable is defaulted to 1'b0 when the class is created and every time the *b_transport()* task is used, it is changed to 1'b1. This serves to inform the other classes when information is available to be treated.

The *b_transport()* field represents the typical task used by socket targets.

On the other hand, the class *socket_master_container* implements the initiator socket. This class is simpler than the class *socket_slave_container*. While the former one implements the *b_transport()* task and the *notification* variable, the *socket_master_container*, besides of creating the initiator socket, only creates an wrapper around the execution of the *b_transport()* task to not have the need to pass the object *uvm_tlm_time*.

Just like *socket_slave_master*, it is necessary to cast the object we want to send into an object of the class *generic_info*. The table 4.9 represents the mentioned class.

socket_slave_container	
Field	Description
Initiator socket	Instantiation of the initiator socket that passes around an object dervied from *generic_info_block*
Variables	Instatiation of the object from *uvm_tlm_time* that is necessary for the execution of *b_transport()* task
Constructor	Typical class constructor
Wrapper for *b_transport()*	Wrapper to take away the need of passing an object from *uvm_tlm_time*
UVM Macros	Typical UVM macros

Table 4.9: Elements of the class *socket_slave_container*

An example for this class usage will follow in code 4.2.

```
1 class i2c_agent_manager extends generic_agent_manager;
2         socket_master_container socket_master    ;
3         generic_info            i2c_info         ;
4 
5         virtual task run_phase(uvm_phase phase);
6                 $cast(generic_info, i2c_info);
7                 socket_master.transport(generic_info);
8         endtask: run_phase
9 endclass: i2c_agent_manager
```

Code 4.2: Code for the usage of socket master container

The file *generic_tb/general_class/sockets.sv* contains the code used to create the socket containers and some examples as well.

4.8 Agents

The classes that represent the agents are highlighted in the figure 4.12.

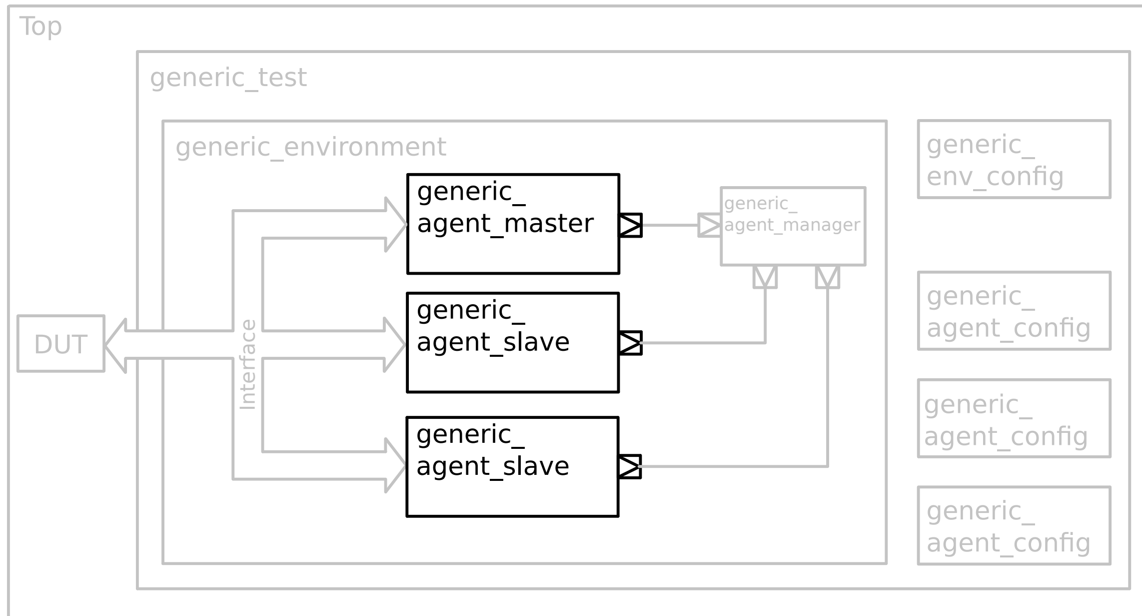

Figure 4.12: Agents of the verification environment

There are two types of agents: master agents and slave agents. The concept of *master* and *slave* refers to the relationship of the components with the agent manager. If a component is classified as master, it means that it will send information to the agent manager, on the other hand if a component is classified as slave, it means that it will receive information from the agent manager.

The master agents are represented by the class *generic_agent_master* and are responsible for sending information about the DUT's state to the agent manager. The slave agents, represented by the class *generic_agent_slave*, are agents controlled by the agent manager on basis with the information provided by the master agents.

The sockets available in the agents, both master and slave, aren't from the classes *socket_*_containers*, they are passthrough sockets, meaning that they just relay the messages to another socket.

4.8.1 Master Agent

A typical composition of a master agent is represented in figure 4.13.

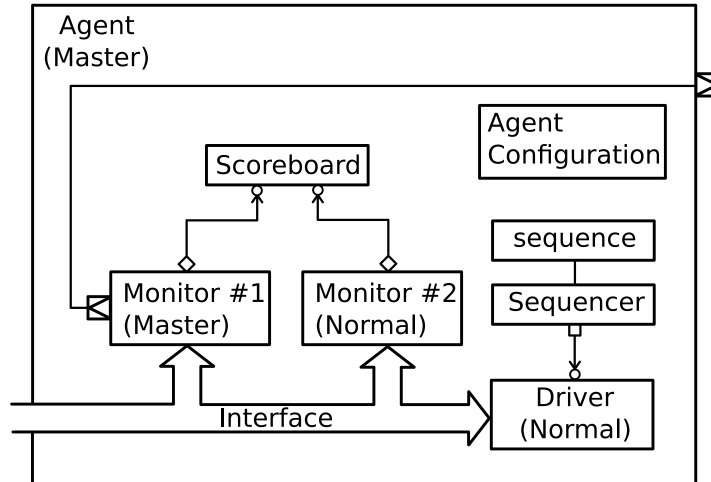

Figure 4.13: A typical constitution of a master agent

The master agent is characterized by having at least one component that sends information to the agent manager through the passthrough socket, usually that component is a master monitor. Master monitors will be seen in detail in the section 4.10.1.

The table 4.10 represents the class *generic_agent_master*.

generic_agent_master	
Field	Description
Socket initiator passthrough	Instantiates a socket passthrough to connect a socket initiator to the socket target of the agent manager
Instantiation of the agent's components	Instatiation of the configuration, sequencers, drivers, etc.
Constructor	Typical class constructor
Build Phase	Creation of the objects of the agent's components
Connect Phase	Connection of the sockets and load the virtual interfaces
Run Phase	Execution of the sequence on the sequencer
UVM Macros	Typical UVM macros

Table 4.10: Elements of the class *generic_agent_master*

The *Socket initiator passthrough* field instantiates the socket passthrough to be used by the agent's components.

The *Instatiation of the agent's components* field instantiates all the necessary agents components (like agent configuration, sequences and sequencers, drivers, monitors and scoreboards).

The *Build Phase* creates the mentioned components and loads the agent's configuration.

The *Connect Phase* connects the socket initiator to the socket passthrough, loads the test configuration into the agent, connects the driver to the sequencer, connects the monitors to the scorboard and loads the interfaces into the monitor and into the drivers.

The *Run Phase* raises the testbench objection and loads a sequence into the sequencer.

4.8.2 Slave Agent

The slave agent is represented by the class *generic_agent_slave* and is very similar to the master agent except in two aspects:

- Instead of master components that send information to the agent manager, it features slave components that receive information from the agent manager
- It features an extra block: the *broadcaster*

The figure 4.14 represents the composition of the slave agent class.

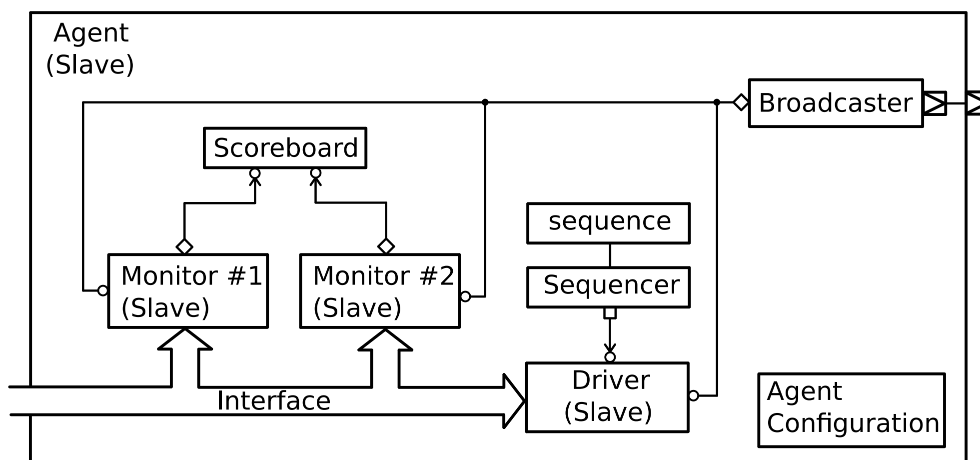

Figure 4.14: A typical constitution of a slave agent

The broadcaster is a block that converts a socket communication into an analysis port, allowing to send the same message to multiple components at the same time. A deeper explanation of the broadcaster can be consulted later in the chapter 4.9.

The components connected to the broadcaster are usually components that are constantly using computing resources which, in the most cases, are the monitors and the drivers. The scoreboards and the sequencers also consume computing resources but they usually depend on the monitors and on the driver to work, so if these are on hold, the scoreboard and the sequencer should be on hold as well.

The class is represented on table 4.11.

generic_agent_slave	
Field	Description
Socket target passthrough	Instantiates a socket passthrough to connect the agent master to the broadcaster
Instantiation of the agent's components	Instatiation of the configuration, sequencers, drivers, etc and of the broadcaster.
Constructor	Typical class constructor
Build Phase	Creation of the objects of the agent's components
Connect Phase	Connection of the sockets and load the virtual interfaces
UVM Macros	Typical UVM macros

Table 4.11: Elements of the class *generic_agent_slave*

The *Socket initiator passthrough* field instantiates the socket passthrough to be used by the brodcaster.

The *Instatiation of the agent's components* field instantiates all the necessary agents components (like agent configuration, sequences and sequencers, drivers, monitors and scoreboards). It also instantiates the broadcaster.

The *Build Phase* creates the mentioned components and loads the agent's configuration.

The *Connect Phase* connects the agent manager to the broadcaster, connects the slave components (like drivers and monitors) to the broadcaster was well, loads the test configuration into the agent, connects the driver to the sequencer, connects the monitors to the scorboard and loads the interfaces into the monitor and into the drivers.

More information can be consulted in the files *generic_tb/agents/generic_agent_master.sv* and *generic_tb/agents/generic_agent_slave.sv*.

4.9 Broadcaster

The broadcaster is a block that converts socket connections into analysis ports. The block will connect to the passthrough socket of the slave agent and it will relay all the messages of the socket to the slave components connected to the analysis port. It is a one-way communication from the socket to the analysis port, which means that the slave components can't send messages back to the agent manager. The broadcaster is derived from the class *uvm_component* in order to be able to use sockets and ports.

The broadcaster block is represented on figure 4.15.

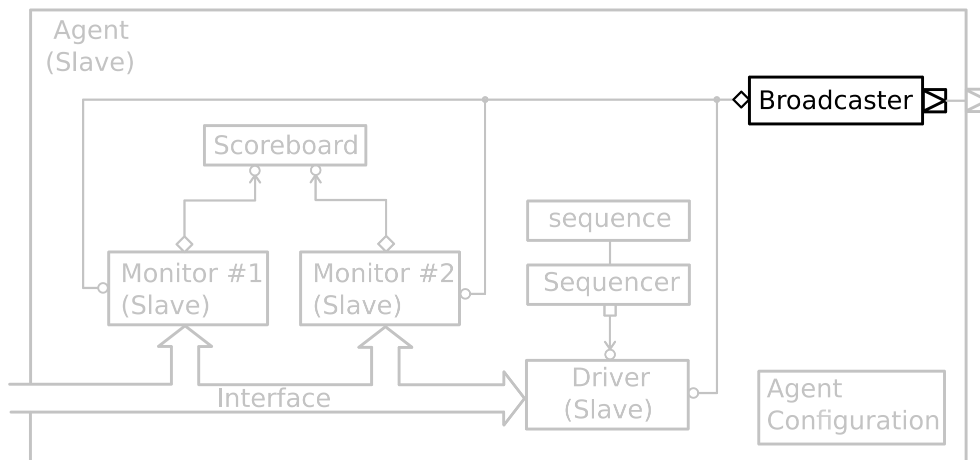

Figure 4.15: The broadcaster block

The composition of the class *broadcaster* is represented on table 4.12.

broadcaster	
Field	Description
Ports and sockets	Instantiates an analysis port and a slave socket
Variables	Instatiation of the object from *generic_info_block* to be transported through the socket to the analysis port
Constructor	Typical class constructor
Build Phase	Creation of the object from *generic_info_block*
Run Phase	Starting of an infinite loop on the task *get_socket_slave()*
Task *get_socket_slave()*	Task to get messages from the socket and to send them to the analysis port
UVM Macros	Typical UVM macros

Table 4.12: Elements of the class *broadcaster*

The "*Ports and socket*" field instantiates the socket to communicate with the agent manager and an analysis port to connect to all the slave components of the agent.

The "*Variables*" field instantiates the object from *generic_info_block* that is collected from the agent manager and sent to the slave components. This block must not be replaced with a derived class or similar. Each component that receives the object *generic_info_block* must cast it to the derived class chosen by the engineer. And the similar happenswith the agent manager, it must cast the derived class to the object *generic_info_block* before sending it through the socket.

The "*Build Phase*" creates the object instantiated in "*Variables*".

The "*Run Phase*" starts an infinite look on the task that gets messages from the agent message and relays them to the analysis port.

The task *get_socket_slave()* listens to the socket's notification variable and waits for it to become active, then it retrives the message from the socket and sends it to the analysis port by calling the task *write()* of each connected port.

The broadcaster is one of the few classes that can be used as it is provided by the testbench, there isn't the necessity of created a child class derived from this one in order to complement it with the rest of the testbench.

The code for this class is located in the file *generic_tb/general_class/broadcaster.sv*.

4.10 Monitors

In this verification environment, there are 3 different types of monitors:

- Master monitors: components derived from the class *generic_monitor_master*
- Slave monitors: components derived from the class *generic_monitor_slave*
- Normal monitors: components derived from the default class *uvm_monitor*

The master monitors are components that listen to the most essential lines of the DUT and gather information about their change of state. They are usually monitors that are responsible for changing and testing the DUT's configuration.

The slave monitors are components that are controlled by the state machine from the agent manager. These monitors receive information blocks derived from *generic_info_block* and they react to that information. As an example, they could stop collecting transactions from the data lines, in case of the functionality, that they are observing, gets disabled by the DUT.

The normal monitors are components isolated from the agent manager, they have no relationship with it.

There might be cases in which deriving classes from *generic_monitor_master* or *generic_monitor_slave* might not make sense. For example, when designing a testbench in which the monitor that collects transactions from the serial bus, that same monitor could be used for both master and slave agents. In this case, it would mean replicating the code through two different classes that derived from *generic_monitor_master* and *generic_monitor_slave*.

So, it is not mandatory to have classes derived from these two generic monitors as long they implement the same features.

4.10.1 Master Monitors

The master monitor is represented in figure 4.16.

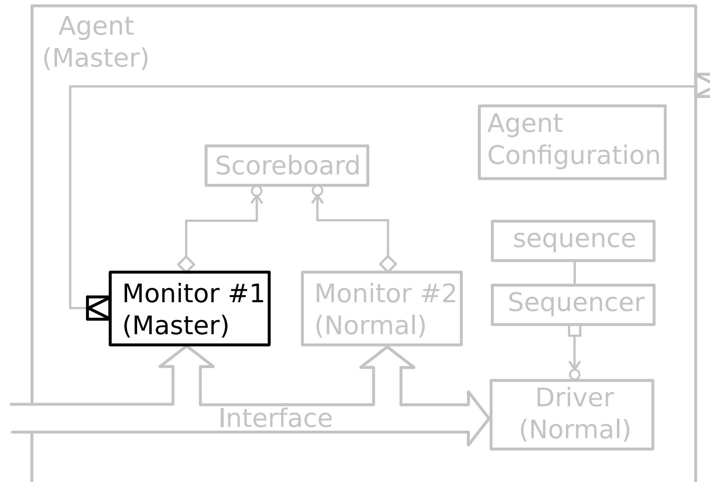

Figure 4.16: The master monitor block

This class does not have the analysis port implemented, it is needed to do it in the derived classes. The composition of the class *generic_monitor_master* is represented on table 4.13.

Table 4.13: Elements of the class *broadcaster*

generic_monitor_master	
Field	Description
Socket	Instantiates master socket
Analysis Port	The analysis port for the communication with the scoreboard is necessary to be instantiated in the derived classes
Virtual Interfaces	For the communication with the DUT
Variables	The object to be sent to the agent manager goes here, as well the variables needed for the monitor
Constructor	Typical class constructor
Run Phase	The tasks that collect transactions from the communication between the DUT and the driver go here
Task *send_to_socket()*	Task to send messages to the agent manager
UVM Macros	Typical UVM macros

The *Socket* field instantiates a master socket container to be used in the task *send_to_socket()*.

The *Analysis Port* field is reserved for the instantiation of the analysis port necessary to send data to the scoreboard. The analysis port must be created in the derived classes.

The *Variables* field is reserved for monitor variables and the instantiation of the object to be sent to the agent manager.

The *Run Phase* represents the main task of the monitor.

The task *send_to_socket()* is the task responsible to submit an object derived from *generic_info_block* to the agent manager. Before executing this task, the object derived from *generic_info_block* must be casted into its parent class. Example: *$cast(generic_info_block, derived_info_block)*

4.10.2 Slave Monitors

A slave monitor is represented in figure 4.17.

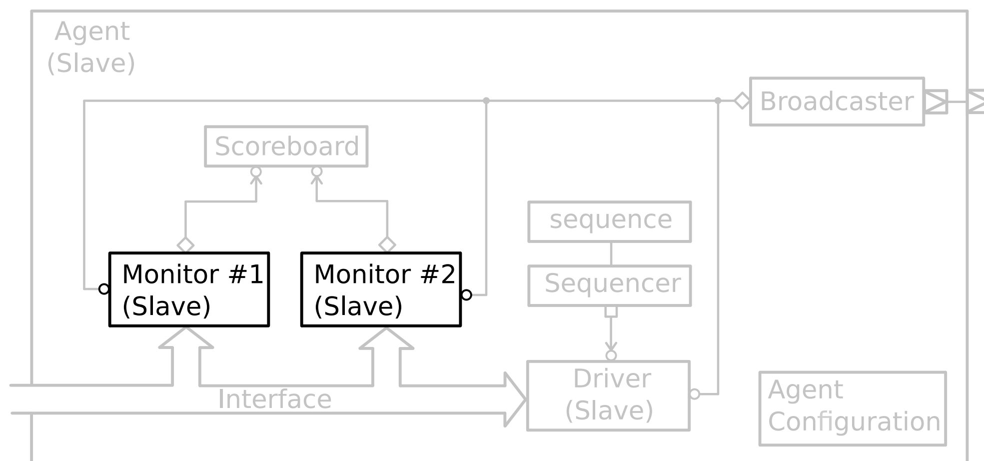

Figure 4.17: The slave slave block

This class does not have the analysis port implemented, it is needed to do it in the derived classes. The composition of the class *generic_monitor_slave* is represented on table 4.14.

Table 4.14: Elements of the class *broadcaster*

generic_monitor_slave	
Field	Description
Analysis export and TLM FIFO	Instantiates an analysis export and a FIFO
Analysis Port	The analysis port for the communication with the scoreboard is necessary to be instantiated in the derived classes
Virtual Interfaces	For the communication with the DUT
Variables	The object to be received from the agent manager goes here, as well the variables needed for the monitor
Constructor	Typical class constructor
Connect Phase	Connects the analysis export to the FIFO
Run Phase	The tasks that collect transactions from the communication between the DUT and the driver go here
Task *send_to_socket()*	Task to send messages to the agent manager
UVM Macros	Typical UVM macros

The *Analysis export and TLM FIFO* instantiates an analysis export, which connects to the analysis port of to the broadcaster, and a FIFO. The FIFO is connected during the *Connect Phase* to the analysis export and it stores all the items received from data received from the broadcaster.

The *Analysis Port* field is reserved for the instantiation of the analysis port necessary to send data to the scoreboard. The analysis port must be created in the derived classes.

The *Variables* field is reserved for monitor variables and the instantiation of the object to be sent to the agent manager.

The *Connect Phase* connects the FIFO to the analysis export.

The *Run Phase* represents the main task of the monitor.

The task *get_fifo()* is the task responsible for getting objects of the class *generic_info_block* from the broadcaster. The information contained on these objects are used to control the operation of the monitor. After the execution of this task, the object of *generic_info_block* must be casted to an object derived from the class *generic_info_block*. Example: *$cast(derived_info_block, generic_info_block)*)

4.10.3 Normal Monitors

In this verification environment, the normal monitors are classified as simple monitors derived from the class *uvm_monitor* that do not have any special feature like embedded sockets or ports.

4.11 Drivers

In a similiar way to the monitors, drivers are divided in two categories:

- Slave drivers: components derived from the class *generic_driver_slave*
- Normal drivers: components derived from the class *uvm_monitor*

The structure and the functionality of the class *generic_driver_slave* is very similar to the structure of the class *generic_monitor_slave*, so the description of section 4.10 is suitable for the class of the current section.

4.12 Scoreboard, Sequencers, sequences and transactions

In an UVM testbench there are three important classes that help to model the communication with the DUT. These are the *sequencer* (usually derived from *uvm_sequencer*), the *sequence* (usually derived from *uvm_sequence*) and the *transaction* (usually derived from *uvm_sequence_item*). These three classes are addressed in better detail in appendix A.5.

The *scoreboard* is a component derived from *uvm_scoreboard* that compares the result of the DUT with the result of the model of the testbench. This component is address in appendix A.9.

Due to their close proximity with the specification of the DUT, these classes are recommended to be designed from the scratch because they contain little information that could be used for this generic verification environment. So they aren't mentioned in the current chapter but they will be mentioned in chapters 5 and 6.

4.13 Work flow

This section will enumerate the steps necessary to build this testbench:

1. Create the virtual interface
   The interface is essential for the communication between the DUT and the testbench, more information about it can be consulted in appendix A.4.

2. Create the top module for the testbench
   The top module will connect the testbench with the DUT and it will save the virtual interface into to the *uvm_config_db*

3. Create the configuration blocks
   The *env* configuration block and the agent configuration block must be created so that they can be instantiated in the *test* class

4. Create the *test* block
   The *test* class will instantiate the configuration blocks from step 3, it will get the virtual interface created in step 1 from *uvm_config_db* and save it in the agent configuration block. The *test* class will also configure any needed directed tests and it will save the agent configuration blocks into the *env* configuration block

5. Create the *env* block
   The configuration block of the *env* is passed to the *env* through the *test* class. The *env* class will pass the agent configuration blocks to each created agent within this class

6. Create the transactions, the sequences and the sequencers
   The transactions and the sequences will represent a model of the communication with the DUT and the sequencer will provide transactions to the driver.

7. Create the drivers
   The drivers are components that interact with the DUT by sending it transactions taken from the sequencers.

8. Create monitors and scoreboard
   The monitors and the scoreboard will provide a mean for testing the DUT.

9. Build an agent using the components created in the previous steps
   The agent should establish the connection between the connections between the monitor and the scoreboards, and between the driver and the sequencer. The agent should also connect the monitors and the driver to the virtual interface. This interface should be present on the agent configuration block. The broadcaster should be created in case of a slave agent.

10. Create the agent manager

11. In the *env* block, establish the connect between the agents and the agent manager

These 11 steps provide a reference point for building a verification environment based on the generic blocks of the current chapter.

4.14 Conclusion

This chapter presented a deep analysis of the developed verification environment. The environment features some of the details mentioned about in sections 3.2 and 3.6 and it should be ready to be applied to real situations.

The chapters 5 and 6 will demonstrate situations in which this testbench can be used. Thes chapters will give continuity to the verification plan of the SOC approached in section 3.6 and it will approach a new device: a model of the audio codec AC97.

Chapter 5

Application of the Environment to the SOC

In chapter 3.4, a group of components were designed to verify I2C interfaces individually. In chapter 3.6 those components were used to design a possible verification environment to accomodate the necessary features of the SOC. In chapter 4 all the components and design decisions of the verification environment were documentated.

This chapter and the chapter 6 will the usage of the environment in two situations: the SOC from chapter 3.5 and a model of an AC97 audio codec, that will be presented later in this thesis.

5.1 Verification of the SOC

The chapter 3.5 presented a device that represented a typical high-speed communciation protocol used by Synopsys. Later in chapter 3.6, it was identified some features and characteristics necessary in order to verify the created device and then, a possible testbench was conceived using the verification components created for I2C interfaces in chapter 3.4.

By applying those components to the verification environment of the chapter 4, an appropriated testbench can be designed to accomodate the functionality of the SOC.

Those components will form three different agents, each agent will connect to each I2C interface. There will be one master agent, *i2c_agent_serialmaster*, along with two slave agents, *i2c_agent_serialslave_1* and *i2c_agent_serialslave_2*, and they will be connected by an agent manager, *i2c_agent_manager*.

5.1.1 I2C Master Agent

The figure 5.1 represents an agent capable of emulating an I2C-Master device. It will be classified as a *master agent* and, as result, it will be derived from *generic_agent_master*, which will be inheriting the passthrough socket for the communication the the agent manager.

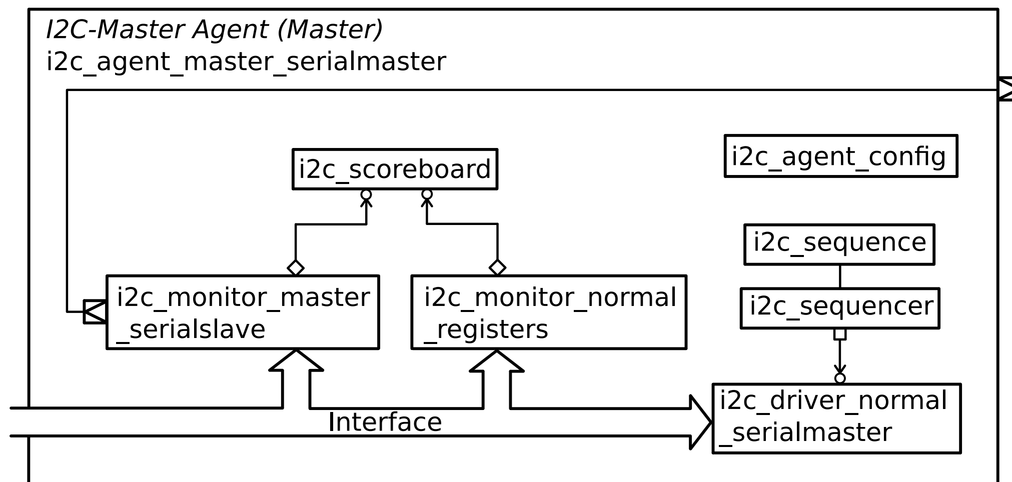

Figure 5.1: Agent for testing an I2C slave interface

The driver and the monitor that will be listening to the parallel port, will not have any relationship with the agent manager, so they are classified as *normal components*. They will be represented by the classes *i2c_driver_normal_serialmaster* and *i2c_monitor_normal_registers*, respectively.

The monitor that will be tapping into the serial line for the communication between the driver and the DUT, will collect transactions about the DUT's change of state, so it will classified as a *master monitor*. As a master monitor, it inherits the socket necessary to send information about the DUT to the agent manager. This monitor will be represented by the class *i2c_monitor_master_serialslave*.

This agent will be used to stimulate the I2C-Slave interface of the SOC, meaning that it will control the functionality of the device. So, the sequencer will generate random transactions that contain: a random slave address, a random read/write bit and a random 8-bit value to be sent to the bus in case of the write operation. As it was mentioned in chapter 3.6, the real address of the slave will be retrieved from the configuration database by the driver and it will be configured through the parallel port of the device.

The slave address, that is sent through an I2C frame, will be randomized by the sequencer. However, there are 127 possible outcomes from this randomization but only one of them will be a match for the slave driver, so there is a probability of $1/127$ in which the sequencer will provide a correct address. This means that most of the tests done will be with an incorrect address.

To avoid this, it is possible to make use of weight distributions in constraints, which is a feature from SystemVerilog [13, p. 145], in order to change the probabilities of the randomized values.

By adding the constraint from code 5.1 to the the transaction, the correct address will have a 50% chance of being generated and an incorrect address will have the remaining 50% chance.

```
1 constraint correct_addr_cons {
2         slave_address dist {real_slave_address:=50,
3                               !(real_slave_address):=50};
4 }
```

Code 5.1: Constraint for balancing the probabilities of generating the real slave address

The variable *real_slave_address* must be added to the transaction and it is value will be defined by the *i2c_sequencer* and by the *i2c_sequence*.

A similar case happens with the data that will be written to the SOC. The SOC only reacts to two possible values: 0x01 (change to low-speed lane) and 0x02 (change to high-spped lane). But the data value has 8 bits and as a result, the probability of these two values being the result of a complete randomization is very tiny. So a similar constraint should be created in order to increase the odds for both desired values. This constraint is represented on code 5.2.

```
1 constraint data_cons {
2         data dist {0x01:=35, 0x02:=35, !(0x01 || 0x02):=30};
3 }
```

Code 5.2: Constraint for balacing the probabilities of generating useful data values

The transaction also implements a *compare_trans(i2c_trans trans)* function. This function accepts another transaction of the same class, it compares the values of each variable and it produces an *uvm_error* in case of mismatched values. This function was created to simplify the functionality of the scoreboard.

The newly created transaction for the master agent is represented on table 5.1. In order to reuse the already exisiting code from the transaction, this transaction was extended from *i2c_trans_master*. So only the variables after the row *New variables* were added.

The sequence is responsible for attributing values to a transaction. The sequence and the sequencer share a relationship in which the sequence can access the variables of the sequencer. The variables of the sequencer can be written by the higher class, which is the agent. This means that the agent can use the information retrieved from the block *i2c_agent_config* to influence how the transactions are generated.

The class *i2c_agent_config* will be derived from *generic_agent_config*, so it will inherit its variables, and it will added the structure mentioned in chapter 3.6. The *test* class will create an object of *i2c_agent_config* and it also define the variable values of the object. The agent will then load this object and it will copy the values of the variables to the sequencer. The sequence can then access to the values from the sequencer and shape the generated transactions.

After the configuration of a configuration object *i2c_agent_config*, the sequence can use the information of this object to enable and disable constraints. In this situation, the class *i2c_sequence* was programmed to take the following options depending on the value of the variable *random*:

Table 5.1: I2C-Master transaction

Transaction: i2c_trans_master (Extends from *i2c_trans*)		
Name	Description	Variable
Slave address	The address to be sent in the I2C frame	logic [6:0] slave_address
Read/write bit	The bit that defines the type of operation	logic rw
Data	The 8 bit message sent through the bus	logic [7:0] data
Number of retries	The number of times that the master tries to communication in case the lack of ACK	int n_retries
New elements		
Actual slave address	The actual address of the slave being tested. This address is sent to the port *addr* of the parallel interface	logic [6:0] real_address
correct_addr_cons	Constraint for balancing the probabilities of generating the real slave address	*N/A*
data_cons	Constraint for balacing the probabilities of generating useful data values	*N/A*
compare_trans(i2c_trans_master)	Function to compare the transaction with another transaction of the same class	*N/A*

- If *random* is equal to 0, the sequence will not randomize the transaction and it will use the values defined in *i2c_agent_config*
- If *random* is equal to 1, the sequence will randomize the transaction but the constraints will be respected
- If *random* is equal to 2, the sequence will do a total randomization of the transaction and it will not take into account the defined constraints

As a result, the values inside a transaction are always dependent of the configuration object even though the object is first created in the *test* class.

The scoreboard is represented by the *i2c_scoreboard* class and it is only responsible for executing the function *compare_trans()* of one of transactions collected by the monitor, by passing the other transaction as an argument. The scoreboard structure is very similar to the one described in appendix A.9.

5.1.2 I2C Slave Agent

The figure 5.2 represents the agent that emulates the functionality of an I2C-Slave device.

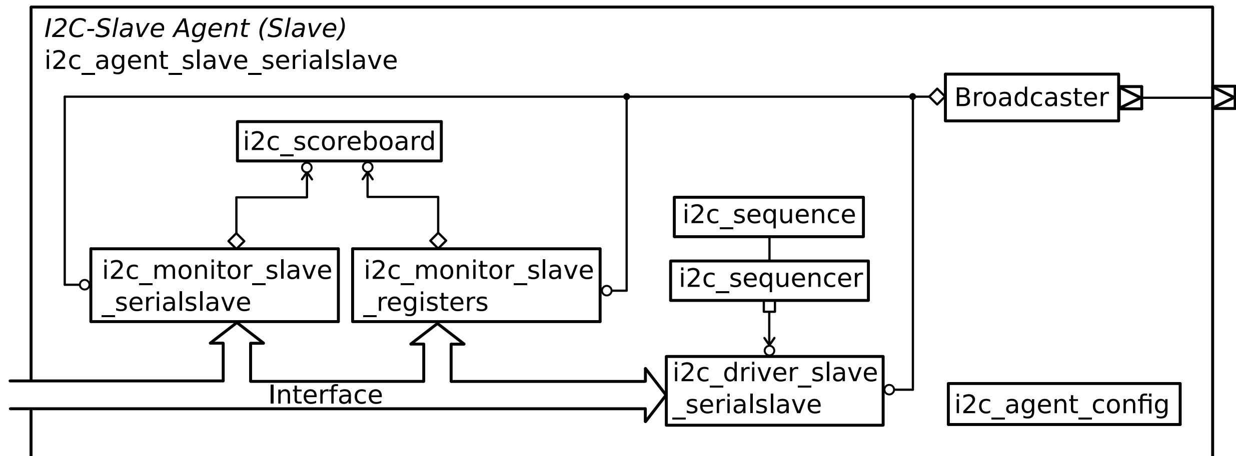

Figure 5.2: Agent for testing an I2C master interface

The structure of this agent and its operation will be very similar to the *i2c_agent_slave_serialmaster*. The driver (*i2c_driver_slave_serialslave*) will maintain activity on the bus, while the monitors (*i2c_monitor_slave_serialslave* and *i2c_monitor_slave_registers*) will be collecting transactions from the serial line and from the parallel interface, so that they can be evaluated by the scoreboard *i2c_scoreboard*. The scoreboard, in its turn, will use the *compare_trans()* available in the transaction to evaluate the test that will be made to the DUT.

But this agent will be distinguished from the master agent due to the broadcaster and the components that will connect to this new block. The driver and to both monitors will be connected to the broascaster to receive information about the DUT's state from the agent manager. These components will cease operation if they receive information that their interface is disabled and they will resume execution if they receive information that their connected interface is enabled. The state machine is based on the the behavior described in section 3.6.3.

5.1.3 Grouping the agents

The complete verification environment is represented in figure 5.3.

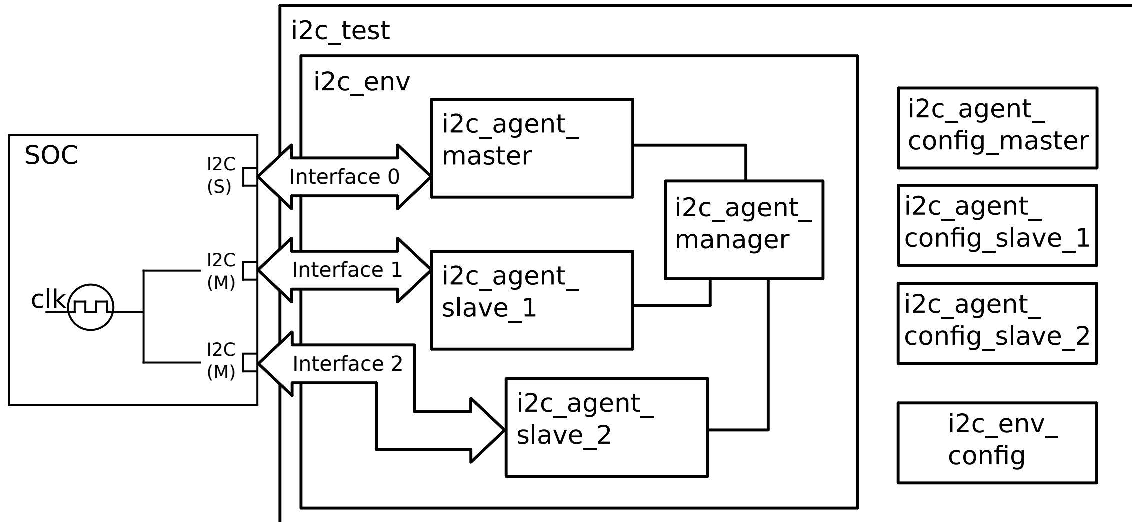

Figure 5.3: Overview of the complete verification environment for the SOC

In order to demonstrate the reconfigurability of the environment, it is considered another revision of the SOC that only features the I2C slave interface because both I2C master interfaces are disabled. To test this new revision, the testbench needs to disable all the slave agents. In order to accomplish this, it is only required to set the variable *agent_active* to 1'b0 in the respective configuration blocks. This example is represented in figure 5.4.

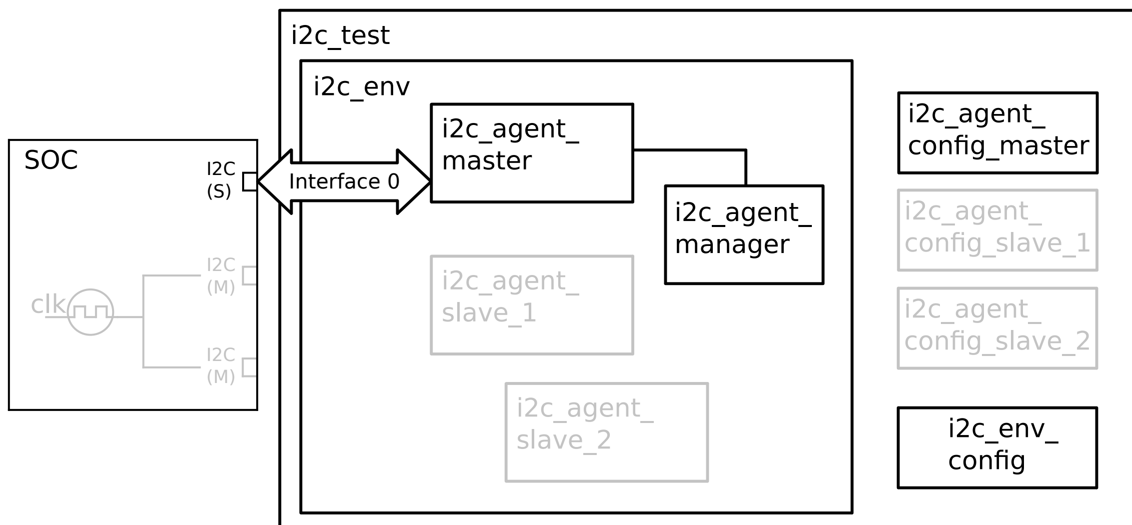

Figure 5.4: Verification environment reconfigured for a revision of the SOC that features only one I2C-Slave

The testbench will proceed then to disconnect both agents from the DUT and from the agent manager as well. The agents will not build the components inside them.

Another example is shown on figure 5.5. The new example describes a revision of the DUT in which is enabled one I2C slave and one I2C master. To build a testbench for this case, it is only required to set the *agent_active* variable of the *ac97_agent_config_master* and of the *ac97_agent_config_slave_2* to 1'b1.

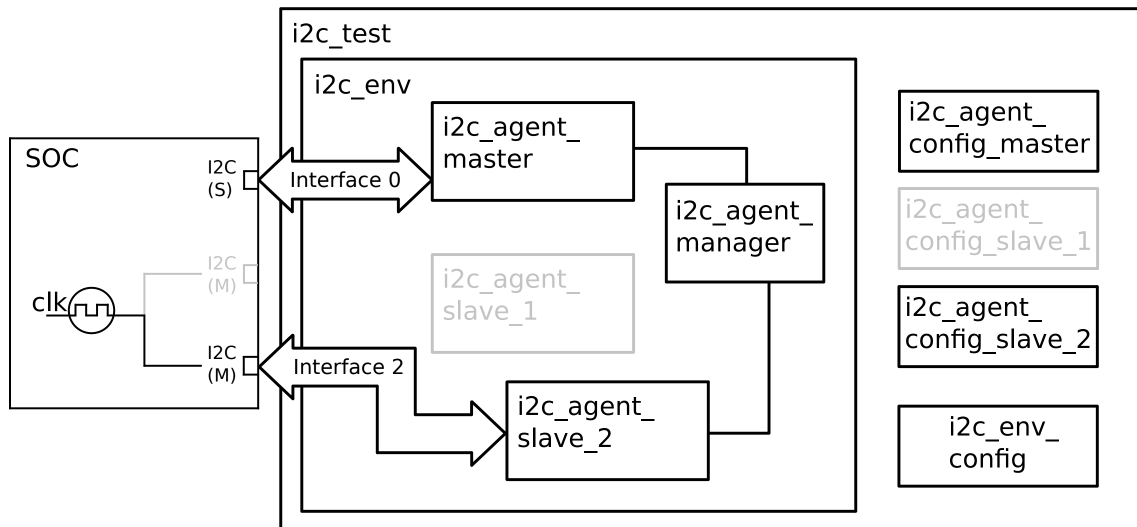

Figure 5.5: Verification environment reconfigured for a revision of the SOC that features one I2C-Slave and one I2C-Master

The presence of the agent manager also allows for the behavior described in 3.6.3. The test-bench will be able to enable and disable slave agents dinamically, depending on the sequences generate by the agent master.

5.2 Conclusion

The verification environment developed on chapter 4 eased the construction of a testbench for the SOC since it established a solid framework in order to build the necessary components on top of it.

Without this framework, the agent manager and the broadcaster would have to be created in order to pass messages along the different agents. The TLM communication available in the generic monitors, drivers and agents would have to be created as well, not to mention the agent manager and the configuration blocks

By having this infrastructure ready to work, the verification engineer can focus its attention to the tasks and functions unique to the device under test.

The table represents the verification components created for the verification environment of the SOC.

Table 5.2: Verification components for the testbench of the SOC

Block type	Block class	Parent class
Monitor I2C-Slave Serial (Master)	i2c_monitor_master_serialslave	generic_monitor_master
Monitor Registers (Normal)	i2c_monitor_normal_registers	uvm_monitor
Monitor I2C-Slave Serial (Slave)	i2c_monitor_slave_serialslave	generic_monitor_slave
Monitor Registers (Slave)	i2c_monitor_slave_registers	generic_monitor_slave
Scoreboard	i2c_scoreboard	uvm_scoreboard
Driver I2C Serial Master	i2c_driver_serialmaster	uvm_driver
Driver I2C Serial Slave	i2c_driver_serialslave	generic_driver_slave
Transaction	i2c_trans	uvm_sequence_item
Transaction	i2c_trans_master	i2c_trans
Sequence	i2c_sequence	uvm_sequence
Sequencer	i2c_sequencer	uvm_sequencer
Scoreboard	i2c_scoreboard	uvm_scoreboard
Agent config	i2c_agent_config	generic_agent_config
Environment config	i2c_env_config	generic_env_config
I2C Info Block	i2c_info_block	generic_info_block
Broadcaster	broadcaster	uvm_component
Agent Manager	i2c_agent_manager	generic_agent_manager
Agent	i2c_agent_master_serialmaster	generic_agent_master
Agent	i2c_agent_slave_serialslave	generic_agent_slave
Env	i2c_environment	generic_environment
Test	i2c_test	uvm_test

Chapter 6

Application of the Environment to the AC97

6.1 Overview of the AC97

After a brief demonstration of the verification environment in section 5.1, a new device will be verified using a testbench derived from the same verification environment. This section will analyse a different type of device to see how the testbench could be implemented in devices that differ from the one created.

The new device under test is an implementation of an AC97 codec, the LM4550 from Texas Instruments. The DUT represents only a partial implementation of this codec and it was developed by the professor José Carlos Alves for one of the courses in Digital System Design, at FEUP.

The model features 2 stereo outputs, 2 stereo inputs, a stereo DAC, a stereo ADC, analog mixers and a serial interface. The model is represented in the figure 6.1.

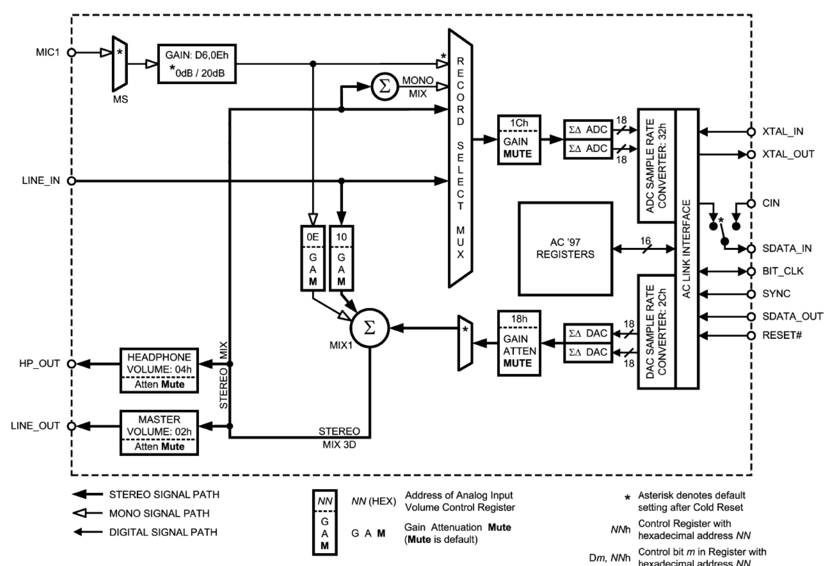

Figure 6.1: A simple model of LM4550 [11, p. 2]

This codec provides a stereo data acquisition system through a dual 18-bit ADC and a stereo analog output from an on-chip stereo DAC. Each output can be muted or attenuated and each mixer has separate gain, attenuation and mute control. It is also possible to feed sound samples through the serial interface and mix them with the input analog signals. These samples are converted with the built-in DAC and the result from the mix can be also be obtain through the serial interface because due to the ADC. [11]

6.1.1 AC-Link Interface

The AC-Link is a synchronous serial link, similar to the SPI interface, that is used to control the codec. It is constituted by five wires: the clock, the *sync* signal, the *reset* and two data wires, *sdata_out* and *sdata_in*. The wire *sdata_in* represents the output of the codec and the wire *sdata_out* represents the input of the codec.

This interface is able to configure the registers responsible for the codec functions (like the multiplexer, the mixer and the gain/mute control) and it is able to transport sound samples to the DAC.

The input and output frames of an AC-Link interface have different structures. The input frame is represented in figure 6.2. [11, p. 18]

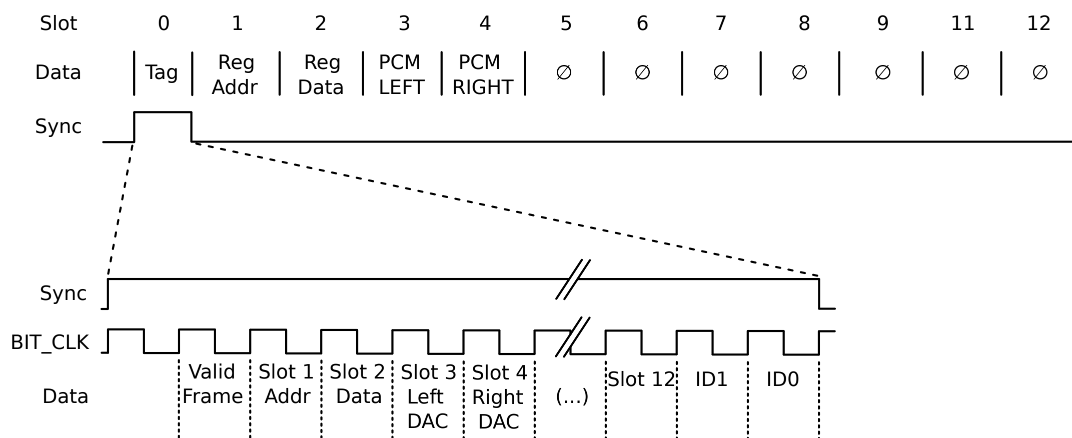

Figure 6.2: Codec Input frame of an AC-Link interface

The input frame is constituted by a Tag phase, 16 bits, and 12 data slots, 20 bits each. In total, each frame consists of 256 bits but in the used model, the last slot is always discarded, so it is only to be considered a frame of 236 bits.

The Tag phase contains information about the validity of the frame and what slots have available data. The data that travels whithin the device is transported in the slots and each slot represents a specific set of data used by the codec.

As it is possible to see in figure 6.2, the first bit of the Tag phase indicates whether the current frame is valid or not, while the next sucession of 12 bits represent each slot and indicate if there is data available in the respective slot. The last two bits, ID0 and ID1, represent the codec address.

The next slots contain the information to be used by the codec. The slot 1 contains the address of the register to be accessed and the slot 2 contains the data that will be written in the address defined by slot 1. The slots 3 and 4 contain the sound samples to be streamed to the codec's DAC, they represent the left and right channels respectively.

Due to the reduced simplicity of the model, only the slots from 0 to 4 are used. Although the data for the other slots is collected by the device, that data is not used.

The available registers are:

- 02h: Master
- 04h: HP_OUT
- 0Eh: MIC1
- 10h: LINE_IN
- 18h: DAC
- 1Ah: Record select mux
- 1Ch: ADC

These registers control the elements of the model represented in figure 6.1 and the values of each of them are defined in the official datasheet. [11, p. 23]

On the other hand, the frame produced by the codec is represented in figure 6.3. [11, p. 20]

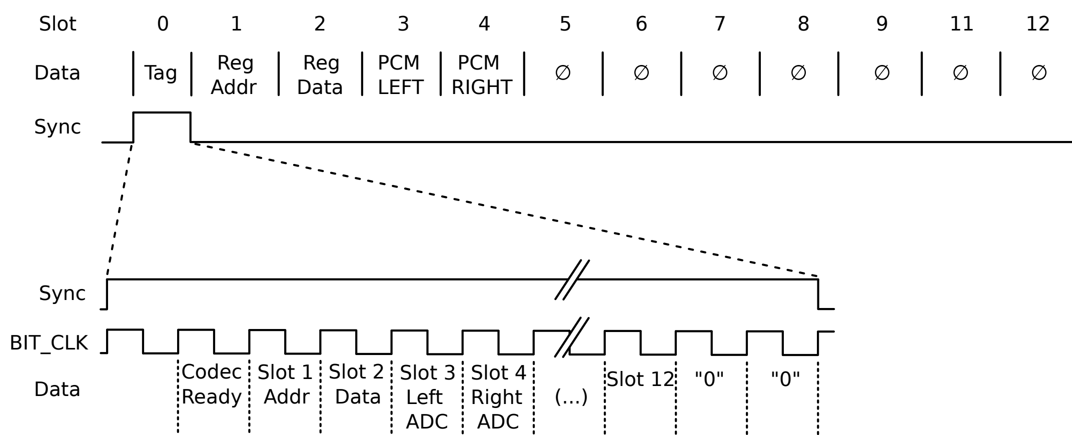

Figure 6.3: Codec Output frame of an AC-Link interface

The output frame is very similar to the input frame, differing in some aspects. The slot 1 represents the 7-bit address sent to the codec through the input frame and the slot 2 represents the data available on that address. Both these slots only are filled with data in case of a read request sent through the input frame. For the example being demonstrated in this chapter, these two slots will not be used in the testbench.

The most important slots are the slots 3 and 4, they represent the output of the codec's ADC. The data present in these two slots will be collected by monitors in the testbench to make an evaluation of the test to be done. The input interface will be accessed by one of the drivers of the testbench in order to configure the device.

The next section (6.2) will describe a verification plan for this device using the developed verification environment.

6.2 Verification components

The DUT is a model of an audio codec, which features audio mixing as main feature. So it is desirable to test 4 main functionalities:

1. First test: AC-Link interface and DUT's registers, section 6.3.1
   The codec is controlled by the AC-Link interface and the registers control the whole device

2. Digital to analog test, section 6.3.2
   The codec is capable of converting digital waveforms into analog waveforms

3. Analog to analog test, section 6.3.3
   The codec is able to mix its analog inputs and send the result to the analog output

4. Analog to digital test, section 6.3.4
   There is the possibility of convert the analog inputs into digital data

The current section will examine the DUT and it will identify which verification components are necessary in order to verify it. After the components for the testbench are defined, the section 6.3 will present the verification plan of the mentioned four possible situations using those components.

6.2.1 Driving the inputs of the AC97 audio codec

By analyzing the figure 6.1 it is possible to identify some important inputs and outputs. There are 3 main inputs:

- One digital input, the *sdata_out*
- Two analog inputs, the *MIC1* and the *LINE_IN*.

In order to generate activity on the DUT, it will be created two types of drivers that will connect to these inputs. One of the drivers will generate a digital frame and it will connect to the signal *sdata_out*, and the other driver will generate an analog waveform and it will connect to the analog inputs.

The *sdata_out* input accepts a frame from the format of the figure 6.2, so the digital driver will take a random transaction of the sequencer and it will generate a similar frame. This driver will be represented by a class denominated of *ac97_driver_normal_aclink_input* and the run phase of the class will consist on a infinite loop, during the simulation, that cycles through the following steps:

1. Request a transaction from the sequencer
2. Generate and sample a waveform to be included in the slots 3 and 4
3. Compose and send an AC-Link input frame

4. Inform the sequencer that the current operation with the transaction is finished and return to step 1

The waveform generator from the step 2 will generate a sawtooth wave with fixed parameters: a sawtooth wave with a frequency of 1 KHz will be generated for the left channel and another sawtooth wave will be generated with a frequency of 2 KHz for the right channel.

On the other hand, the register address and the respective data will be obtained by a random transaction from the sequencer. So, these two variables can be added to a transaction named *ac97_trans_base*.

This transaction will represent the communication from the testbench to the DUT, so it will contain the variables for the register's address and the data to be written on the DUT. It will also contain a constraint to limit the values to the registers available in the codec. A function to compare the transaction with another one of the same class was also created under the name of *compare_regs(ac97_trans_base trans)*.

The table 6.1 represents the transaction *ac97_trans_base*.

Table 6.1: Transaction for generating values for the sine generator - *ac97_trans_base*

Transaction: ac97_trans_base		
Name	Description	Variable
Register's address	Address of the register that it is intended to write	logic [6:0] regaddr
Data	Data to be written at the register's address	logic [15:0] data
Registers bank	Copy of the bank of registers of the DUT	logic [15:0] regs []
DAC Left	Data for the left channel of the DAC	logic [17:0] dac_left
DAC Right	Data for the right channel of the DAC	logic [17:0] dac_right
constraint regaddr_cons	Constraint to limit the addresses generated in this transaction	*N/A*
compare_regs()	Function to compare the transaction with another transactions of the same class	*N/A*

The code for the constraint can be seen in code 6.1.

```
1 constraint regaddr_cons{
2         regaddr inside {7'h1A, 7'h18, 7'h1C, 7'h0E,
3                         7'h10, 7'h04, 7'h02};
4 }
```

Code 6.1: Constraint for limiting the addresses randomized in *ac97_trans_base*

Besides of a transaction, a sequencer and a sequence generator will also be necessary, so the classes *ac97_sequencer_regs* and *ac97_sequence_regs* will be created as well.

The second driver will only be active when the analog inputs are in use by the DUT, so it will be derived from *generic_driver_slave* and it will be represented by a class named *ac97_driver_slave_-sine*. This driver will generate a sine waveform with a frequency and attenuation randomized by the transaction *ac97_trans_sine* and it will cycle through these steps:

1. Create a parallel process that generates a sinewave for LINE_IN (left and right channels) and MIC1

2. Request a transaction from the sequencer and update the frequency values of the sinewave generators

3. Inform the sequencer that the operation of the transaction is done and return to step 2

The transaction *ac97_trans_sine* is represented on table 6.2.

Table 6.2: Transaction for generating values for the sine generator - *ac97_trans_sine*

Transaction: ac97_trans_sine		
Name	Description	Variable
Frequency Left	Frequency of the left channel	logic [15:0] freq_left
Frequency Right	Frequency of the right channel	logic [15:0] freq_right
Attenuation Left	Attenuation of the left channel	logic [15:0] attenuation_left
Attenuation Right	Attenuation of the right channel	logic [15:0] attenuation_right

The transaction is obtained through *ac97_sequencer_sine* and generated with *ac97_sequence_-sine*, which can force values to the transaction or generate random values.

6.2.2 Collecting data items from the AC97's inputs and outputs

While the drivers pulls transactions from the sequencer to maintain activity on the digital and analog inputs, the testbench will need components that listen to the same activity. When it comes to the digital interface, two monitors will be created:

- A monitor that listens to the parallel interface of the DUT to check the result of the driver's influence: *ac97_monitor_normal_parallel_regs*

- A monitor that listens to the activity on the bus and that emulates the DUT's functionality: *ac97_monitor_master_aclink_input*

The first monitor checks the parallel interface to access directly to the DUT's registers bank and observess how the DUT reacts to the transactions from the driver *ac97_driver_normal_aclink_input*. It is classified as a *normal monitor* because it does not have any influence in the agent manager nor it is controlled by it. The monitor will send the accessed address and the respective data to a scoreboard to be evaluated by the testbench.

The second monitor will listen to the AC-Link interface and it will emulate the DUT with the information that is sent through the *sdata_out* wire, and it will collect transactions for the scoreboard about the accessed registers. This monitor will be considered a *master monitor*, as it will hold important information about the DUT's state, and it will belong to a class named *ac97_monitor_master_aclink_input*. The agent manager will receive the emulated registers bank every time a new transaction is collected from this monitor.

Along with register's data, it is also sent through the wire *sdata_out*, digital waveforms to be processed by the DAC. In order to create a different test for this functionality, it will be created another monitor that collects information about the those waveforms through the serial interface. The monitor will only be active if there is a change in the DUT's registers that enables both the DAC and the analog outputs, so it will be classified as a *slave monitor*. It will be represented by the class *ac97_monitor_slave_aclink_dac* which will be derived from *ac97_monitor_slave_aclink_base*.

The class *ac97_monitor_slave_aclink_base* contains the core functionality of the AC-Link interface and the necessary structure to be controlled by the agent manager. This class will be derived to another monitor too, the *ac97_monitor_slave_aclink_adc*, which will be used to get results from the ADC through the wire *sdata_in*. This monitor will store the collected values to the variables *dac_left* and *dac_right* of the class *ac97_trans_base*.

To recap, until now, four different monitor were created to collect the necessary data from the digital IOs. In order to collect transactions from the analog IOs, two another monitors will be created:

1. A monitor from the class *ac97_monitor_slave_parallel_analog_input*
2. A monitor from the class *ac97_monitor_slave_parallel_analog_output*

Both of these monitors are derived from *ac97_monitor_slave_parallel_analog_base*, which contains some common functions that they share, like the function for creating the transaction and sending the transaction to the scoreboard, and also the analysis port. They are classified as *slave monitors* because they will only be active when the codec enables the ports they are using.

The monitor *ac97_monitor_slave_parallel_analog_input* connects to the inputs *MIC1* and *LINE_IN* and emulates the core functionality of the codec, which means that this monitor has a reference model of the codec, a *golden model*. This golden model will emulate the functionality of the codec and it will send the result to a scoreboard.

Meanwhile, the monitor *ac97_monitor_slave_parallel_analog_output* will observe the analog outputs, *LINE_OUT* and *HP_OUT*, collect their values into transactions and send the result to the same scoreboard as *ac97_monitor_slave_parallel_analog_input*.

The transaction that will be collected by both these monitors will be a transaction of the class *ac97_trans_ana*. It will not be used to generate data items for drivers components, it will only be used for collecting samples of the codec's outputs.

The transaction *ac97_trans_ana* is represented on table 6.3.

Table 6.3: Transaction for collecting transactions from the codec's outputs - *ac97_trans_ana*

Transaction: ac97_trans_ana		
Name	Description	Variable
LINE_OUT Left channel	Output of the Master left channel	logic [63:0] line_out_li
LINE_OUT Right channel	Output of the Master right channel	logic [63:0] line_out_ri
HP_OUT Left channel	Output of the Headphone left channel	logic [63:0] hp_out_li
HP_OUT Right channel	Output of the Headphone right channel	logic [63:0] hp_out_ri
compare_ana()	Function to compare the transaction with another transaction of the same class	*N/A*

6.2.3 Evaluating the results of the test

The scoreboard is a very simple component. The evaluation functions are already implemented in the transactions themselves, so the main task of the scoreboard is limited to receiving the transactions from the monitors and executing the *compare_*()* functions. Each test will be represented by an agent and each agent will have its own scoreboard.

6.2.4 Agent manager

The agent manager will be represented by the class *ac97_agent_manager*. This class will control the four tests described in the beginning of section 6.2 and each of these tests will be represented by an agent. The section 6.3 will build these agents, and in section 6.3.5 it will be explained how the state machine available in the agent manager will behave.

6.2.5 Summary of the verification components

The table 6.4 presents a list of all the base classes created for this testbench.

Table 6.4: Elements of the AC97 testbench

Block type	Block class	Parent class
Transaction Registers	ac97_trans_base	*uvm_sequence_item*
Sequence Registers	ac97_sequence_regs	*uvm_sequence*
Sequencer Registers	ac97_sequencer_regs	*uvm_sequencer*
Transaction Sine	ac97_trans_sine	*uvm_sequence_item*
Sequence Sine	ac97_sequence_sine	*uvm_sequence_item*
Sequencer Sine	ac97_sequencer_sine	*uvm_sequencer*
Transaction Analog Output	ac97_trans_ana	*ac97_trans_base*
Driver AC-Link Input (Normal)	ac97_driver_normal_aclink_input	*uvm_driver*
Driver Sine (Slave)	ac97_driver_slave_sine	*generic_driver_slave*
Monitor AC-Link Input (Master)	ac97_monitor_master_aclink_input	*generic_master_monitor*
Monitor Parallel Regs (Normal)	ac97_monitor_normal_parallel_regs	*uvm_monitor*
Monitor AC-Link Base (Slave)	ac97_monitor_slave_aclink_base	*generic_slave_monitor*
Monitor AC-Link Input DAC (Slave)	ac97_monitor_slave_aclink_dac	*ac97_monitor_slave_aclink_base*
Monitor AC-Link Output ADC (Slave)	ac97_monitor_slave_aclink_adc	*ac97_monitor_slave_aclink_base*
Monitor Analog Base (Slave)	ac97_monitor_slave_parallel_analog_base	*generic_slave_monitor*
Monitor Analog Input (Slave)	ac97_monitor_slave_parallel_analog_input	*ac97_monitor_slave_parallel_analog_base*
Monitor Analog Output (Slave)	ac97_monitor_slave_parallel_analog_output	*ac97_monitor_slave_parallel_analog_base*
Scoreboard	ac97_scoreboard	*uvm_scoreboard*
Agent Manager	*ac97_agent_manager*	*generic_agent_manager*

6.3 Test cases

The last section provided an overview of the verification components created on top of the developed verification environment. The next chapters will apply these components to a full featured testbench in order to cover the four test cases enumerated in chapter 6.2.

6.3.1 First test: Testing the DUT's registers

The first test case that it is desirable verify is the registers bank of the codec together with the AC-Link input interface, since these two elements will be controlling the whole functionality of the device. The figure 6.4 represents the functionality of the DUT that will be tested.

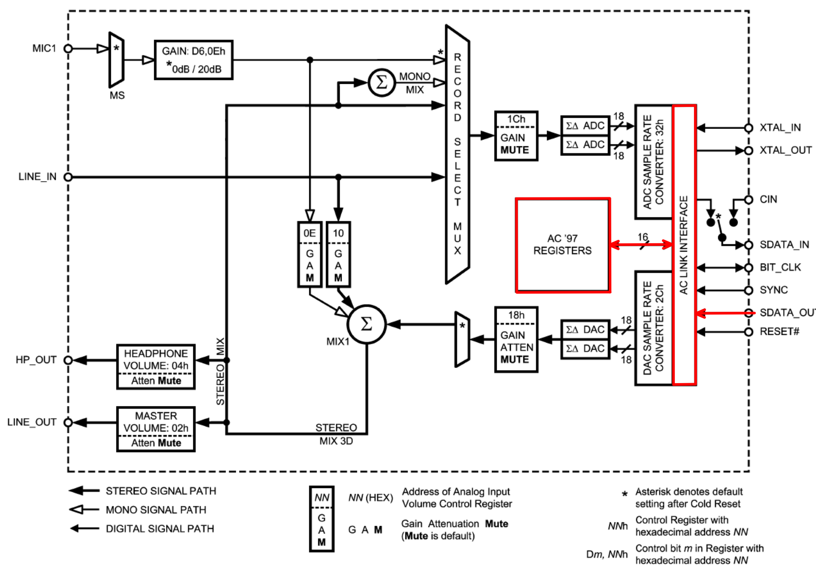

Figure 6.4: LM4550 registers highlighted

Ideally, the testbench would send a random register through the AC-Link interface and it would check the impact on the registers of the codec. At the same time, the testbench would listen to the same interface and emulate the expected behavior. Finally, it would compare that same behavior with the modifications made to the registers and it would look for any mismatches between the DUT's behavior and the the emulated behavior. If they match, the test would pass, otherwise, it would be that something wrong.

A question may arise from the previous paragraph: what if the problem it is in the testbench instead of the DUT? That could happen as well but what is expected for from a verification process is an agreement between the DUT and the testbench. This is one of the reasons that design and verification is done by different teams, the design team developes the device itself while the verification team develops a model of the same functionality of the DUT, a *golden model*. This process reduces redudancy in the interpretation of the specification.

Using the verification components from section 6.2, a testbench can be built for this test case (figure 6.5).

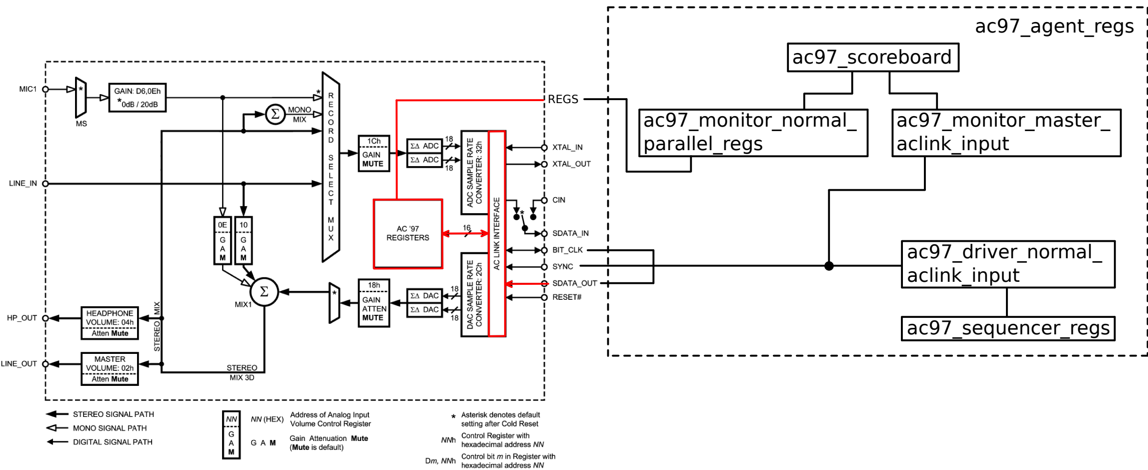

Figure 6.5: LM4550 Testbench for the registers

The components presented here work as described in section 6.2. They all form a single component, the *ac97_agent_regs*. This agent is responsible for testing the interface AC-Link along with the DUT's registers and it will be classified as an agent master, as it will inform the agent manager about the state of the DUT.

6.3.2 Second test: Testing the Digital to Analog functionality

After building a test case for testing the registers, the audio mixing functionality of the codec can start to be tested. The first analog test to be realized will be the conversion digital to analog, which is highlighted in figure 6.6.

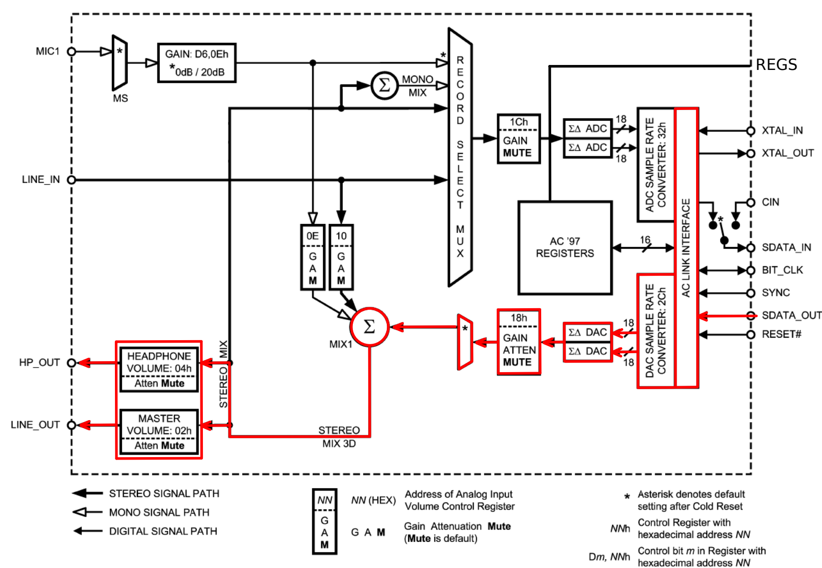

Figure 6.6: LM4550 digital to analog functionality highlighted

The digital waveform that is sent through the AC-Link interface is generated by the *ac97_driver-_normal_aclink_input* of the agent *ac97_agents_regs*, so it will not be needed a driver for this test. The test will consist on two monitors, one of them (*ac97_monitor_slave_aclink_dac*) will tap into the AC-Link input interface to collect the values to be inputed in the DAC, and the other one (*ac97_monitor_slave_parallel_analog_output*) will tap into the analog outputs.

The monitor *ac97_monitor_slave_aclink_dac* has included a golden model of the DUT in order to successfully emulate the functionality of the DAC.

The new agent is represented in figure 6.7.

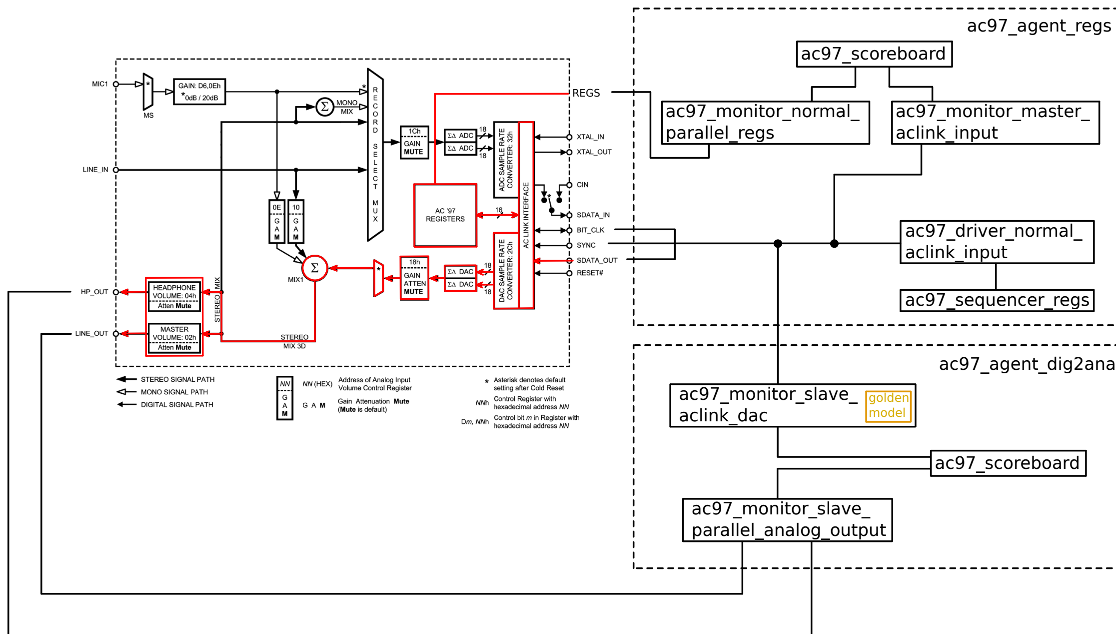

Figure 6.7: LM4550 Testbench for the digital to analog functionality

The agent is represented by the class *ac97_agent_dig2ana* and it is classified as a *slave agent*. This test will be controlled by *ac97_agent_regs* and it will only be operational when both DAC and the analog outputs are enabled by the *ac97_agent_regs*. The association between the agents *ac97_agent_regs* and *ac97_agent_dig2ana* is established by the class *ac97_agent_manager*.

The relationship between agents it is represented in figure 6.8.

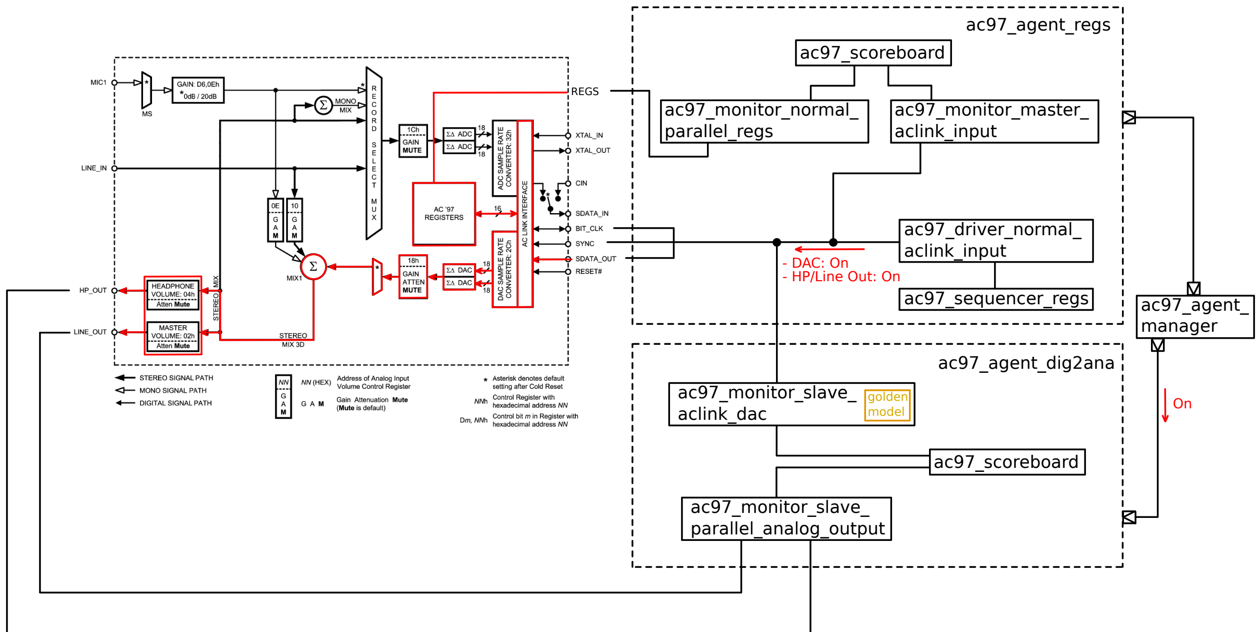

Figure 6.8: LM4550 Testbench for the digital to analog functionality with the agent manager

The agent manager will be connected to both slave monitors of the second agent, these monitors will only be active when, both, DAC and analog ouputs are enabled. When disabled, the monitors will not produce any test.

6.3.3 Third test: Testing the Analog to Analog functionality

Another test that needs to be made is the analog to analog mixing. The functionality is represented in figure 6.9.

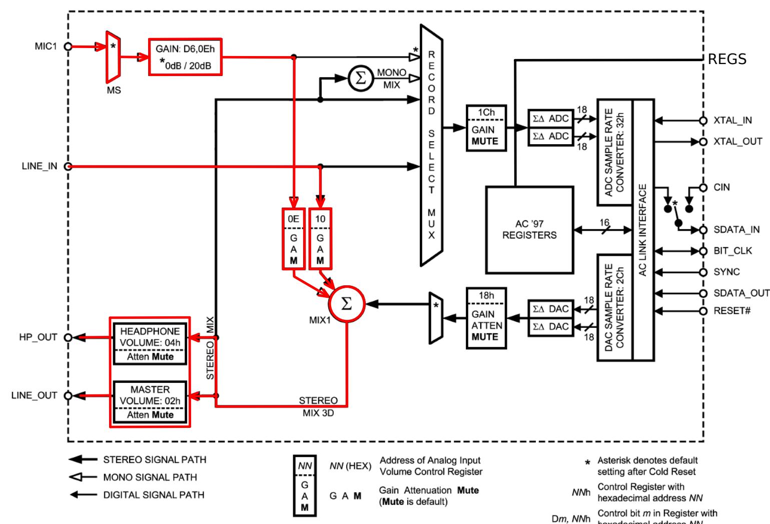

Figure 6.9: LM4550 analog to analog functionality highlighted

This agent belongs to the class *ac97_agent_ana2ana*, which features a *broadcaster* to relay messages from the agent manager to the components responsible to execute the test. The driver and the monitor receive these messages and if they contain information that the outputs are unmuted, they components start executing the test.

The agent is represented in figure 6.10.

Figure 6.10: LM4550 Testbench for the analog to analog functionality

The driver (*ac97_driver_slave_analog_input*) generates sine waves depending on the frequency and attenuation values of the transactions generated by the sequencer. The monitor *ac97_monitor_slave_parallel_analog_input* will then collect the waveforms generated by the driver and emulate the functionality of the codec. Meanwhile the monitor *ac97_monitor_slave_parallel_analog_output* will check the output of the DUT and send the results to the scoreboard.

It is worth noticing that for this test, the agent *ac97_agent_dig2ana* is not enabled. This is the result of the DAC being disabled, rendering the agent unusable. The state machine of the *ac97_agent_manager* is described in section 6.2.

As it is possible to see, the agent manager is able to dinamically add and remove lanes and agents.

6.3.4 Fourth test: Testing the Analog to Digital functionality

The last test case is responsible for testing the analog to digital functionality. This functionality is represented on figure 6.11.

Figure 6.11: LM4550 analog to digital functionality highlighted

The agent that represents this test belongs to the class *ac97_agent_ana2dig*. It is a slave agent, so it features a broadcaster as well. But, just like the agent *ac97_agent_dig2ana*, it does not have a driver included, it will listen instead to the output of the driver of *ac97_agent_ana2ana*.

The agent is represented on figure 6.12.

Figure 6.12: LM4550 Testbench for the analog to digital functionality

This agent is dependable of the *ac97_agent_ana2ana* to operate, this is due to the lack of the driver. With just a few lines of code, the analog driver and the respective sequencer could be easily

added to the current agent, but this situation was chosen in order to demonstrate an example in which agents may depend from resources of other agents.

The situation is dealt normally by the agent manager, as long as its state machine is programmed with this aspect in mind. Whenever the test analog to digital needs to be made, the agent manager can send a message to the *ac97_agent_ana2ana* to execute the necessary resource, in this case, the driver. The other monitors of the same agent can remain disabled while the driver operates. Although, both monitors and the driver receive the same kind of message, they can be programmed to interpret it differently.

6.3.5 Automatization of the environment

The agent manager has the purpose of automate the testbench. This means that, instead of disabling and enabling agents manually every time a different test has to be made, the agent can take charge of this operation.

This functionality is based on a state machine which follows the following rules:

- If the input mixers are mutted, and the DAC and the analog output *HP_OUT*, or *LINE_OUT*, are unmuted, the agent *ac97_agent_dig2ana* will be enabled. Otherwise it woill be disabled.
- If, at least, one of the input mixers, *MIC1* or *LINE_IN* is unmuted, as well with at least one of the analog outputs, the agent *ac97_agent_ana2ana* will be enabled. Otherwise it will be disabled.
- If the DAC is muted, the ADC unmuted, the *Record Select Mux* set to one of the four available paths and, at least, one of the input mixers are unmuted, the agent *ac97_agent_ana2dig* will be enabled. Otherwise it will be disabled.

It is worth noting that this verification plan is not intended to be a complete and thorough plan, it is only intended to demonstrate the variety of situations that could be used by the developed verification environment

6.4 Conclusion

This chapter analyzed a DUT that differed from the category of communication protocols mentioned in chapter 3. This was done with the intention of showing how the developed testbench would behave in a device that diverged from the SOC developed in section 3.5.

Once again, the testbench proved to be convenient by offering the basic infrastructure and guidelines in order to verify a variety of situations. Just like the SOC, the main features of the audio codec were divided in different tests and different agents were created to execute each one of those tests. The agent manager was responsible to manage all the tests, on base of the information provided by the master agent of this DUT.

However, for this device, the agent manager revealed to be more useful than the testbench with the SOC. In the fourth test case, the agent manager allowed to enable and disable individual

resources of a different agent without activating an undesirable test. This situation shows the flexibility of the verification environment created for this dissertation.

Chapter 7

Conclusion

This chapter presents an overview of all the work accomplished during this dissertation, along with an analysis of the objectives that were initially assigned to the project.

7.1 Summary of the developed work

The main objective of this project was the development of a generic verification environment in SystemVerilog by following the UVM methodology. The verification environment had to take into consideration the attributes of high-speed communication protocols developed by Synopsys, such as the possibility of multiple configurations, the presence of multiple lanes and the behavior that their share between them.

In order to accomplish this project, multiple phases were planned:

- The first phase consisted in the acquisition of concepts of SystemVerilog, of functional verification and of UVM. Afterwards, the work was proceeded to the analysis of one of the existing verification environments used by Synopsys in order to identify the desirable features for a generic testbench
- The second phase consisted in designing a device that would accomodate some of the features studied earlier and the respective testbench
- The third phase was destined to separating the elements of the testbench that could be reused to other devices and develop a new verification environment
- The final phase proposed an implementation of the developed environment to, not only, the device designed specially for this dissertation (the SOC) but also to a model of an audio codec AC97. The application of the environment to the audio codec was intended to show the versatility of the developed environment.

This document demonstrated the adaptability and the usefulness of verification methodologies. A well designed, and documented, verification environment can be reused and reconfigured to

different devices, helping to reduce the set up time for a verification process. This was the main reason for the deep and thorough explanation of every each class in chapter 4, The usage of the environment should be accompanied, not only, with this document but, also, with the comments available in the source code.

In the website created for this dissertation, it is possible to find a learning guide for UVM. This guide was not originally planned for this dissertation but it was developed in parallel with this project, mostly due to the lack learning resources of UVM available publicly.

The guide is intended to aid people, with no knowledge in UVM, in giving the first steps in this methodology. It provides the basic understanding of each class and an explanation how to succesfully compile a complete environment. The guide can also be consulted in the appendix A.

7.2 Features and results of the concluded work

As it is possible to conclude from chapters 6 and 5, the verification environment achieved the expected results. It was able to successfully deal with two different devices: one of them was created specially for this dissertation and the other one was based on a already exisiting device.

In the end, the testbench is capable of:

- Activating and deactivating agents from the testbench manually
- Establishing relationship master-slave between the agents
- Dinamically, managing active agents in the testbench (this mean, to automatically change each agent's state, like, enabling and disabling them)
- Activating individual resources of an agent so that they can be used by another one (it is demonstrated in section 6.3.4)
- Supporting multiple configurations of the same DUT by activating and deactivating agents
- Forcing directed tests through configuration blocks

Building verification environments from scratch present a great challenge. However, during the conception of a digital electronic device, verification is almost as important as the design of the device itself, so it should not be neglected.

The SystemVerilog language and the UVM methodology are recent technologies that were created assist the process of verifying digital. Hopefully, the cases, the testbench and the guide studied, developed and written during this dissertation, may help other people to get started in verification methodologies to further improve the quality of their projects.

Appendix A

UVM Guide for Beginners

This guide was developed in parallel with the dissertation and it was created from the need to contribute for more resources about UVM. The guide is inserted as an appendix in order to help to clarify some points about UVM that weren't explained in the main document. Some of the parts written from the guide were reused for the dissertation, so it is normal to see some duplicated excerpts.

The code for this tutorial can be found in the website: http://colorlesscube.com/

A.1 Introduction

As digital systems grow in complexity, verification methodologies get progressively more essential. While in the early beginnings, digital designs were verified by looking at waveforms and performing manual checks, the complexity we have today don't allow for that kind of verification anymore and, as a result, designers have been trying to find the best way to automate this process.

The SystemVerilog language came to aid many verification engineers. The language featured some mechanisms, like classes, covergroups and constraints, that eased some aspects of verifying a digital design and then, verification methodologies started to appear.

UVM is one of the methodologies that were created from the need to automate verification. The Universal Verification Methodology is a collection of API and proven verification guidelines written for SystemVerilog that help an engineer to create a robust verification environment. It's an open-source standard maintained by Accellera and can be freely acquired in their website.

By mandating a universal convention in verification techniques, engineers started to develop generic verification components that were portable from one project to another, this promoted the cooperation and the sharing of techniques among the user base. It also encouraged the development of verification components generic enough to be easily extended and improved without modifying the original code.

All these aspects contributed for a reduced effort in developing new verification environments, as designers can just reuse testbenches from previous projects and easily modify the components to their needs.

This document will provide a training guide for verifying a basic adder block using UVM. The guide will assume that you have some basic knowledge of SystemVerilog and will require accompaniment of the following resources:

- Accellera's UVM User's Guide 1.1:
  http://www.accellera.org/downloads/standards/uvm/uvm_users_guide_1.1.pdf
- Accellera's UVM 1.1 Class Reference:
  http://www.accellera.org/downloads/standards/uvm/UVM_1.1_Class_Reference_Final_06062011.pdf
- Verification Academy's UVM Cookbook:
  https://verificationacademy.com/cookbook/uvm
- Book "SystemVerilog for Verification: A Guide to Learning the TestBench Language Features", Chris Spear
- Book "Comprehensive Functional Verification: The Complete Industry Cycle" by John Goss

This guide will be divided in 3 different parts:

1. The first part, starting on chapter 2, will explain the operation of the device under test (DUT): the inputs, the outputs and the communication bus
2. The second part, starting on chapter 3, will give a brief overview of a generic verification environment and the approach into verifying the DUT
3. The third part, starting on chapter 4, will start to describe a possible UVM testbench to be used with our DUT with code examples. It's important to consult to the external material in order to better understand the mechanism behind the testbench.

A.2 The DUT

This training guide will focus on showing how we can build a basic UVM environment, so the device under test was kept very simple in order to emphasize the explanation of UVM itself.

The DUT used is a simple ALU, limited to a single operation: the add operation. The inputs and outputs are represented in the figure A.1.

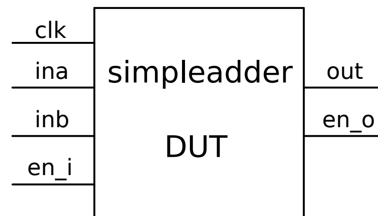

Figure A.1: Representation of the DUT's inputs/outputs

This DUT takes two values of 2 bits each, *ina* and *inb*, sums them and sends the result to the output *out*. The inputs are sampled to the signal of *en_i* and the output is sent at the same time *en_o* is signalled.

The operation of the DUT is represented as a timing diagram and as a state machine in the figure A.2.

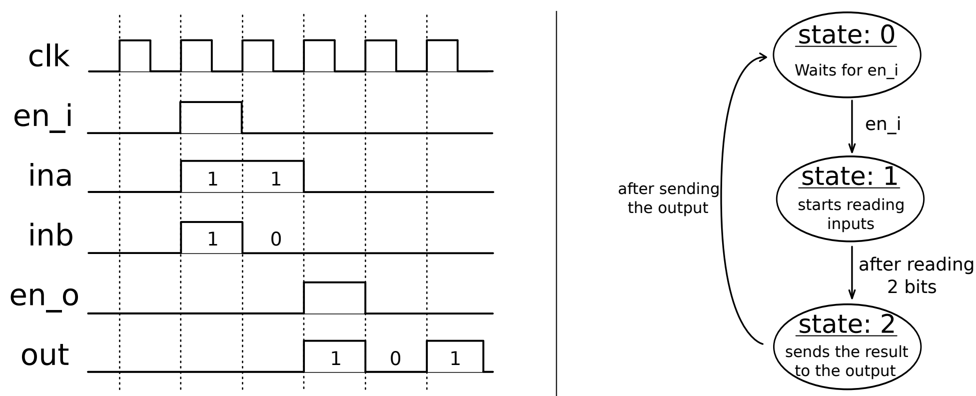

Figure A.2: Operation of the DUT

A.3 Defining the verification environment

Before understanding UVM, we need to understand verification.

Right now, we have a DUT and we will have to interact with it in order to test its functionality, so we need to stimulate it. To achieve this, we will need a block that generates sequences of bits to be transmitted to the DUT, this block is going to be named *sequencer*.

Usually sequencers are unaware of the communication bus, they are responsible for generating generic sequences of data and they pass that data to another block that takes care of the communication with the DUT. This block will be the *driver*.

While the driver maintains activity with the DUT by feeding it data generated from the sequencers, it doesn't do any validation of the responses to the stimuli. We need another block that listens to the communication between the driver and the DUT and evaluates the responses from the DUT. This block is the *monitor*.

Monitors sample the inputs and the outputs of the DUT, they try to make a prediction of the expected result and send the prediction and result of the DUT to another block, the *scoreboard*, in order to be compared and evaluated.

All these blocks constitute a typical system used for verification and it's the same structure used for UVM testbenches.

You can find a representation of a similar environment in the figure A.3.

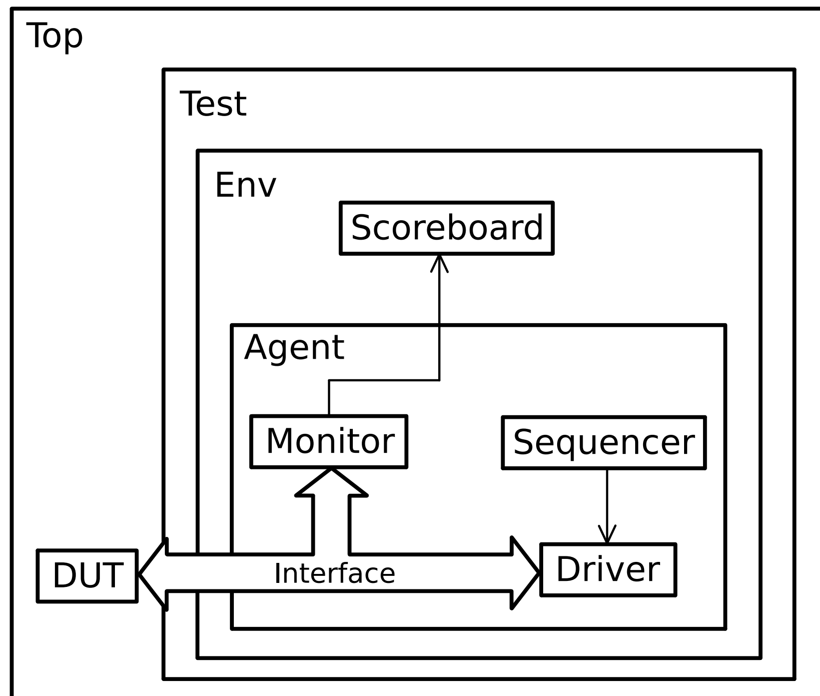

Figure A.3: Typical UVM testbench

Usually, sequencers, drivers and monitors compose an *agent*. An agent and a scoreboard compose an *environment*. All these blocks are controlled by a greater block denominated of *test*. The *test* block controls all the blocks and subblocks of the testbench. This means that just by changing a few lines of code, we could add, remove and override blocks in our testbench and build different environments without rewriting the whole test.

To illustrate the advantage of this feature, let's imagine a situation where we are testing a another DUT that uses SPI for communication. If, by any chance, we want to test a similar DUT but with I2C instead, we would just need to add a monitor and a driver for I2C and override the existing SPI blocks, the sequencer and the scoreboard could reused just fine.

A.3.1 UVM Classes

The previous example demonstrates one of the great advantages of UVM. It's very easy to replace components without having to modify the entire testbench, but it's also due to the concept of classes and objects from SystemVerilog.

In UVM, all the mentioned blocks are represented as objects that are derived from the already existent classes.

A class tree of the most important UVM classes can be seen in the figure A.4.

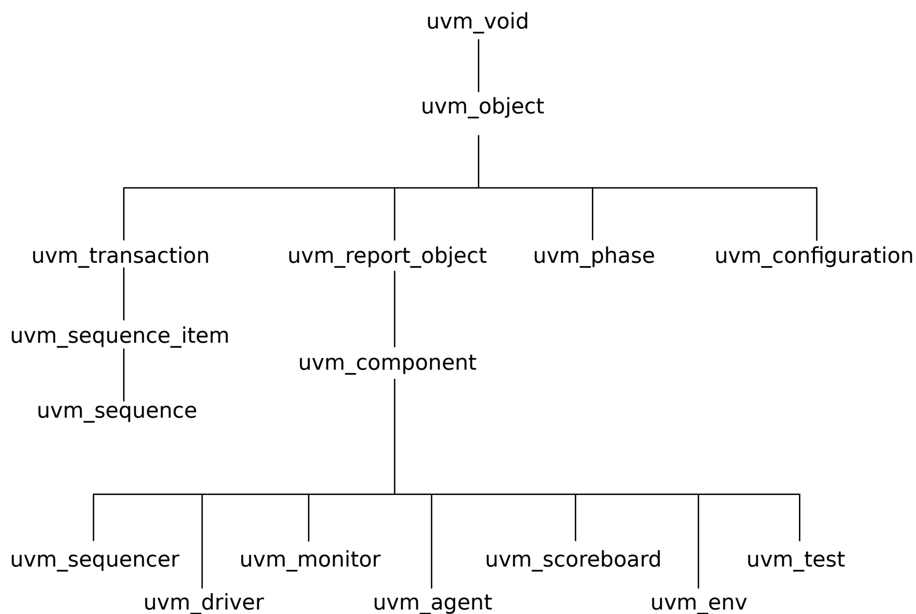

Figure A.4: Partial UVM class tree

The data that travels to and from our DUT will stored in a class derived either from *uvm_sequence_item* or *uvm_sequence*. The sequencer will be derived from *uvm_sequencer*, the driver from *uvm_driver*, and so on.

Every each of these classes already have some useful methods implemented, so that the designer can only focus on the important part, which is the functional part of the class that will verify the design. These methods are going to addressed further ahead.

For more information about UVM classes, you can consult the document Accellera's UVM 1.1 Class Reference.

A.3.2 UVM Phases

All these classes have simulation phases. Phases are ordered steps of execution implemented as methods. When we derive a new class, the simulation of our testbench will go through these different steps in order to construct, configure and connect the testbench component hierarchy.

The most important phases are represented in the figure .

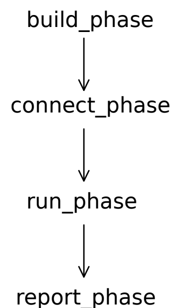

Figure A.5: Partial list of UVM phases

A brief explanation of each phase will follow:

- The build phase is used to construct components of the hierarchy. For example, the build phase of the agent class will construct the classes for the monitor, for the sequencer and for the driver.
- The connect is used to connect the different sub components of a class. Using the same example, the connect phase of the agent would connect the driver to the sequencer and it would connect the monitor to an external port.
- The run phase is the main phase of the execution, this is where the actual code of a simulation will execute.
- And at last, the report phase is the phase used to display the results of the simulation.

There are many more phases but none of them are mandatory. If we don't need to have one in a particular class, we can just omit it and UVM will ignore it.

More information about UVM phasing can be consulted in Verification Academy's UVM Cookbook, page 48.

A.3.3 UVM Macros

Another important aspect of UVM are the macros. These macros implement some useful methods in classes and in variables. they are optional, but recommended.

The most common ones are:

- ‘uvm_component_utils - This macro registers the new class type. It’s usually used when deriving new classes like a new agent, driver, monitor and so on.
- ‘uvm_field_int - This macro registers a variable in the UVM factory and implements some functions like copy(), compare() and print().
- ‘uvm_info - This a very useful macro to print messages from the UVM environment during simulation time.

This guide will not go into much detail about macros, their usage is always the same for every class, so it’s not worth to put much thought into it for now.

More information can be found in Accellera’s UVM 1.1 Class Reference, page 405.

A.3.4 Typical UVM class

All this said, a typical UVM class will look a lot like the one described in the code A.1.

```
class generic_component extends uvm_component;
        `uvm_component_utils(generic_component)

        function new(string name, uvm_component parent);
                super.new(name, parent);
        endfunction: new

        function void build_phase(uvm_phase phase);
                super.build_phase(phase);

                //Code for constructors goes here
        end_function: build_phase

        function void connect_phase(uvm_phase phase);
                super.connect_phase(phase);

                //Code for connecting components goes here
        endfunction: connect_phase

        task run_phase(uvm_phase phase);
                //Code for simulation goes here
        endtask: run_phase

        function void report_phase(uvm_phase phase);
                //Code for showing simulation results goes here
        endfunction: report_phase
endclass: generic_component
```

Code A.1: Code for a generic component

The code listed here, is the most basic sample that all components will share as you will see from now on.

A.3.5 SimpleAdder UVM Testbench

After a brief overview of a UVM testbench, it's time to start developing one. By the end of this guide, we will have the verification environment from the the figure A.6.

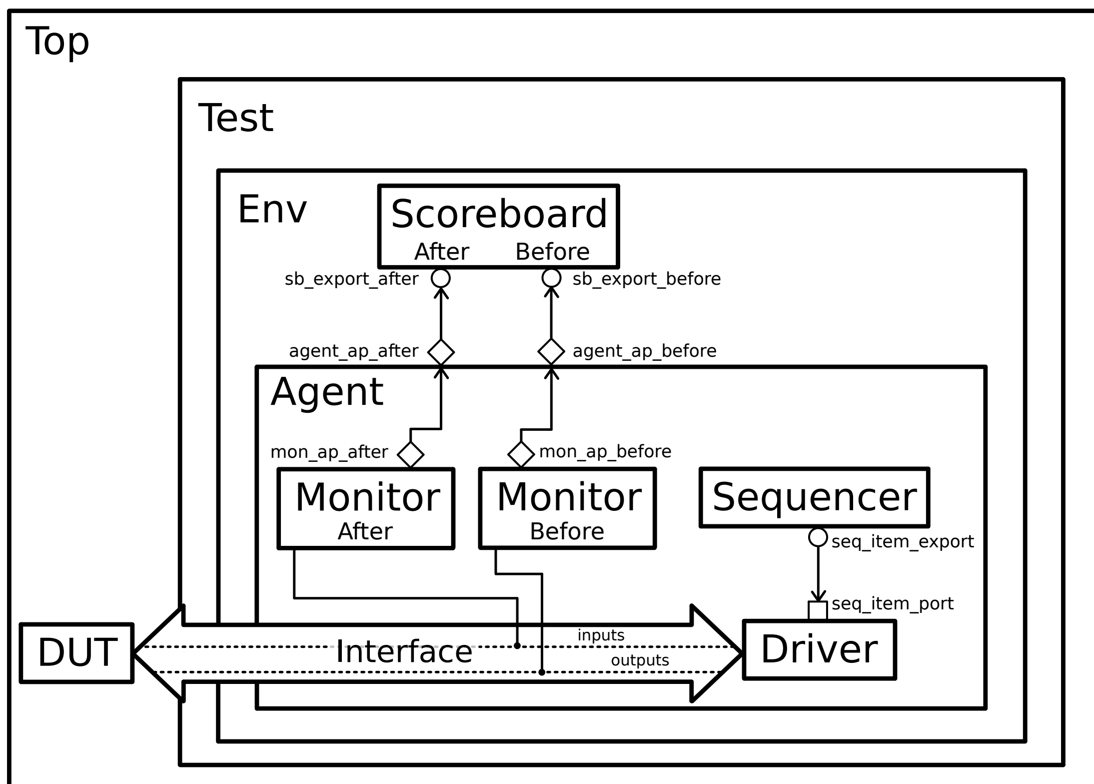

Figure A.6: SimpleAdder Final Testbench

This guide will begin to approach the *top block* and the *interface* (chapter 4), then it will explain what data will be generated with the *sequences* and *sequencers* on chapter 5.

Following the sequencers, it will explain how to drive the signals into the DUT and how to observe the response in chapters 6 and 7 respectively.

Subsequently, it will explain how to connect the sequencer to the driver and the monitor to the scoreboard in chapter 8. Then it will show to build a simple *scoreboard* in chapter 9.

And finally, the test will be executed and its output analyzed.

For the complete code of the testbench, it is provided a Makefile used to run the simulation. This Makefile uses Synopsys' VCS but it should be easily modifiable to be executed with any HDL simulator.

A.4 Top Block

In a normal project, the development of the DUT is done separately from the development of the testbench, so there are two components that connects both of them:

- The top block of the testbench
- A virtual interface

The top block will create instances of the DUT and of the testbench and the virtual interface will act as a bridge between them.

The interface is a module that holds all the signals of the DUT. The monitor, the driver and the DUT are all going to be connected to this module.

The code for the interface can be seen in the code A.2.

```
interface simpleadder_if;
        logic   sig_clock;
        logic   sig_ina;
        logic   sig_inb;
        logic   sig_en_i;
        logic   sig_out;
        logic   sig_en_o;
endinterface: simpleadder_if
```

Code A.2: Interface module - simpleadder_if.sv

After we have an interface, we will need the top block. This block will be a normal SystemVerilog module and it will be responsible for:

- Connecting the DUT to the test class, using the interface defined before.
- Generating the clock for the DUT.
- Registering the interface in the UVM factory.
  This is necessary in order to pass this interface to all other classes that will be instantiated in the testbench.
  It will be registered in the UVM factory by using the *uvm_resource_db* method and every block that will use the same interface, will need to get it by calling the same method.
  It might start to look complex, but for now we won't need to worry about it too much.
- Running the test.

The source for the top block is represented in the code A.3.

```
`include "simpleadder_pkg.sv"
`include "simpleadder.v"
`include "simpleadder_if.sv"

module simpleadder_tb_top;
	import uvm_pkg::*;

	//Interface declaration
	simpleadder_if vif();

	//Connects the Interface to the DUT
	simpleadder dut(vif.sig_clock,
			vif.sig_en_i,
			vif.sig_ina,
			vif.sig_inb,
			vif.sig_en_o,
			vif.sig_out);
	initial begin
		//Registers the Interface in the configuration block
		//so that other blocks can use it
		uvm_resource_db#(virtual simpleadder_if)::set
			(.scope("ifs"), .name("simpleadder_if"), .val(vif));

		//Executes the test
		run_test();
	end

	//Variable initialization
	initial begin
		vif.sig_clock <= 1'b1;
	end

	//Clock generation
	always
		#5 vif.sig_clock = ~vif.sig_clock;
endmodule
```

Code A.3: Top block - simpleadder_tb_top.sv

A brief explanation of the code will follow:

- The lines 2 and 3 include the DUT and the interface into the top block, the line 5 imports the UVM library, lines 11 to 16 connect the interface signals to the DUT.
- Line 21 registers the interface in the factory database with the name *"simpleadder_if"*.
- Line 25 runs one of the test classes defined at compilation runtime. This name is specified in the Makefile.

- Line 35 generates the clock with a period of 10 timeunits. The timeunit is also defined in the Makefile.

For more information about interfaces, you can consult:

- Book "SystemVerilog for Verification: A Guide to Learning the TestBench Language Features", chapter 5.3

A.5 Transactions, sequences and sequencers

The first step in verifying a RTL design is defining what kind of data should be sent to the DUT. While the driver deals with signal activities at the bit level, it doesn't make sense to keep this level of abstraction as we move away from the DUT, so the concept of transaction was created.

A transaction is a class object, usually extended from *uvm_transaction* or *uvm_sequence_item* classes, which includes the information needed to model the communication between two or more components.

Transactions are the smallest data transfers that can be executed in a verification model. They can include variables, constraints and even methods for operating on themselves. Due to their high abstraction level, they aren't aware of the communication protocol between the components, so they can be reused and extended for different kind of tests if correctly programmed.

An example of a transaction could be an object that would model the communication bus of a master-slave topology. It could include two variables: the address of the device and the data to be transmitted to that device. The transaction would randomize these two variables and the verification environment would make sure that the variables would assume all possible and valid values to cover all combinations.

In order to drive a stimulus into the DUT, a so-called driver component converts transactions into pin wiggles, while a so-called monitor component performs the reverse operation, converting pin wiggles into transactions.

After a basic transaction has been specified, the verification environment will need to generate a collection of them and get them ready to be sent to the driver. This is a job for the *sequence*. Sequences are an ordered collection of transactions, they shape transactions to our needs and generate as many as we want. This means if we want to test just a specific set of addresses in a master-slave communication topology, we could restrict the randomization to that set of values instead of wasting simulation time in invalid values.

Sequences are extended from *uvm_sequence* and their main job is generating multiple transactions. After generating those transactions, there is another class that takes them to the driver: the sequencer. The code for the sequencer is usually very simple and in simple environments, the default class from UVM is enough to cover most of the cases.

A representation of this operation is shown in the figure A.7.

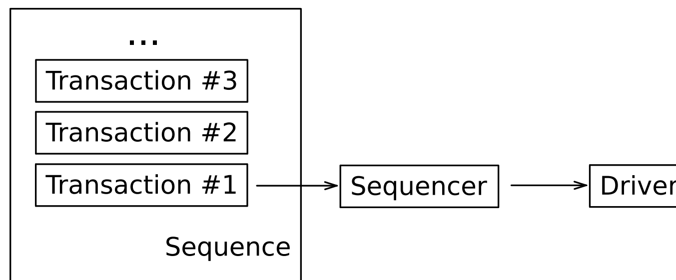

Figure A.7: Relation between a sequence, a sequencer and a driver

The sequence englobes a group of transactions and the sequencer takes a transaction from the sequence and takes it to the driver.

To test our DUT we are going to define a simple transaction, extended from *uvm_sequence_item*. It will include the following variables:

- rand bit[1:0] ina
- rand bit[1:0] inb
- bit[2:0] out

The variables *ina* and *inb* are going to be random values to be driven to the inputs of the DUT and the variable *out* is going to store the result. The code for the transaction is represented in the code A.4.

```
class simpleadder_transaction extends uvm_sequence_item;
        rand bit[1:0] ina;
        rand bit[1:0] inb;
        bit[2:0] out;

        function new(string name = "");
                super.new(name);
        endfunction: new

        `uvm_object_utils_begin(simpleadder_transaction)
        `uvm_field_int(ina, UVM_ALL_ON)
        `uvm_field_int(inb, UVM_ALL_ON)
        `uvm_field_int(out, UVM_ALL_ON)
        `uvm_object_utils_end
endclass: simpleadder_transaction
```

Code A.4: Transaction for the simpleadder

An explanation of the code will follow:

- Lines 2 and 3 declare the variables for both inputs. The *rand* keyword asks the compiler to generate and store random values in these variables.
- Lines 6 to 8 include the typical class constructor.
- Lines 10 to 14 include the typical UVM macros.

These few lines of code define the information that is going to be exchanged between the DUT and the testbench.

To demonstrate the reuse capabilities of UVM, let's imagine a situation where we would want to test a similar adder with a third input, a port named *inc*.

Instead of rewriting a different transaction to include a variable for this port, it would be easier just to extend the previous class to support the new input.

It's possible to see an example in the code A.7.

```
class simpleadder_transaction_3inputs extends simpleadder_transaction;
        rand bit[1:0] inc;

        function new(string name = "");
                super.new(name);
        endfunction: new

        `uvm_object_utils_begin(simpleadder_transaction)
        `uvm_field_int(inc, UVM_ALL_ON)
        `uvm_object_utils_end
endclass: simpleadder_transaction_3inputs
```

Code A.5: Extension of the previous transaction

As a result of the class *simpleadder_transaction_3inputs* being an extension of *simpleadder_transaction*, we didn't need to declare again the other variables. While in small examples, like this one, this might not look like something useful, for bigger verification environments, it might save a lot of work.

A.5.1 Sequence

Now that we have a transaction, the next step is to create a sequence.

The code for the sequencer can be found in the code A.6.

```
class simpleadder_sequence extends uvm_sequence#(simpleadder_transaction);
	`uvm_object_utils(simpleadder_sequence)

	function new(string name = "");
		super.new(name);
	endfunction: new

	task body();
		simpleadder_transaction sa_tx;

		repeat(15) begin
			sa_tx = simpleadder_transaction::type_id::create(...

			start_item(sa_tx);
			assert(sa_tx.randomize());
			finish_item(sa_tx);
		end
	endtask: body
endclass: simpleadder_sequence
```

Code A.6: Code for the sequencer

An explanation of the code will follow:

- Line 8 starts the task *body()*, which is the main task of a sequence.
- Line 11 starts a cycle in order to generate 15 transactions
- Line 12 initializes a blank transaction
- Line 14 is a call that blocks until the driver accesses the transaction being created
- Line 15 triggers the *rand* keyword of the transaction and randomizes the variables of the transaction to be sent to the driver
- Line 16 is another blocking call which blocks until the driver has completed the operation for the current transaction

A.5.2 Sequencer

The only thing missing is the sequencer. The sequence will be extended from the class *uvm_sequencer* and it will be responsible for sending the sequences to the driver. The sequencer gets extended from *uvm_sequencer*. The code can be seen in the code A.7.

```
typedef uvm_sequencer#(simpleadder_transaction) simpleadder_sequencer;
```

Code A.7: Extension of the previous transaction

The code for the sequencer is very simple, this line will tell UVM to create a basic sequencer with the default API because we don't need to add anything else.

The structure of the environment is represented on figure A.8.

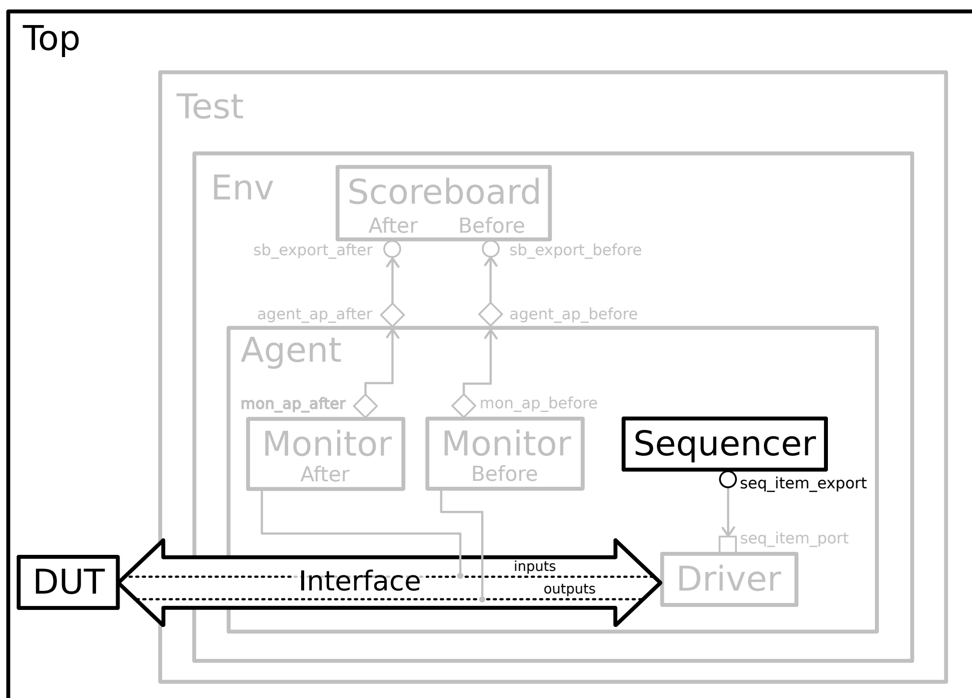

Figure A.8: State of the verification environment after the sequencer

You might have noticed two things missing:

1. How does the sequence connects to the sequencer?
2. How does the sequencer connects to the driver?

The connection between the sequence and the sequencer is made by the *test* block, we will come to this later on chapter 11, and the connection between the sequencer and the driver will be explained on chapter 8.

For more information about transactions and sequences, you can consult:

- Accellera's UVM 1.1 User's Guide, page 48
- Verification Academy's UVM Cookbook, pages 188 and 200

A.6 Driver

The driver is a block whose role is to interact with the DUT. The driver pulls transactions from the sequencer and sends them repetitively to the signal-level interface. This interaction will be observed and evaluated by another block, the monitor, and as a result, the driver's functionality should only be limited to send the necessary data to the DUT.

In order to interact with our adder, the driver will execute the following operations: control the *en_i* signal, send the transactions pulled from the sequencer to the DUT inputs and wait for the adder to finish the operation.

So, we are going to follow these steps:

1. Derive the driver class from the *uvm_driver* base class
2. Connect the driver to the signal interface
3. Get the item data from the sequencer, drive it to the interface and wait for the DUT execution
4. Add UVM macros

In the code A.8 you can find the base code pattern which is going to be used in our driver.

```
class simpleadder_driver extends uvm_driver#(simpleadder_transaction);
	`uvm_component_utils(simpleadder_driver)

	//Interface declaration
	protected virtual simpleadder_if vif;

	function new(string name, uvm_component parent);
		super.new(name, parent);
	endfunction: new

	function void build_phase(uvm_phase phase);
		super.build_phase(phase);
		void'(uvm_resource_db#(virtual simpleadder_if)::read_by_name
			(.scope("ifs"), .name("simpleadder_if"), .val(vif)));
	endfunction: build_phase

	task run_phase(uvm_phase phase);
		//Our code here
	endtask: run_phase
endclass: simpleadder_driver
```

Code A.8: Driver component - simpleadder_driver.sv

The code might look complex already but what it's represented it's the usual code patterns from UVM. We are going to focus mainly on the *run_phase()* task which is where the behavior of the driver will be stated. But before that, a simple explanation of the existing lines will be given:

- Line 1 derives a class named simpleadder_driver from the UVM class uvm_driver.

  The *#(simpleadder_transaction)* is a SystemVerilog parameter and it represents the data type that it will be retrieved from the sequencer.

- Line 2 refers to the UVM utilities macro explained on chapter 2.

- Lines 7 to 9 are the class constructor.

- Line 11 starts the build phase of the class, this phase is executed before the run phase

- Line 13 gets the interface from the factory database. This is the same interface we instantiated earlier in the top block.

- Line 17 is the run phase, where the code of the driver will be executed.

Now that the driver class was explained, you might be wondering: "What exactly should I write in the run phase?"

Consulting the state machine from the chapter 2, we can see that the DUT waits for the signal *en_i* to be triggered before listening to the *ina* and *inb* inputs, so we need to emulate the states 0 and 1. Although we don't intend to sample the output of the DUT with the driver, we still need to respect it, which means, before we send another sequence, we need to wait for the DUT to output the result.

To sum up, in the run phase the following actions must be taken into account:

1. Get a sequence item

2. Control the *en_i* signal

3. Drive the sequence item to the bus

4. Wait a few cycles for a possible DUT response and tell the sequencer to send the next sequence item

The driver will end its operation the moment the sequencer stops sending transactions. This is done automatically by the UVM API, so the designer doesn't need to to worry with this kind of details.

In order to write the driver, it's easier to implement the code directly as a normal testbench and observe its behavior through waveforms. As a result, in the next subchapter (6.0.8), the driver will first be implemented as a normal testbench and then we will reuse the code to implement the run phase.

A.6.1 Creating the driver as a normal testbench

For our normal testbench we will use regular Verilog code. We will need two things: generate the clock and idesginate an end for the simulation. A simulation of 30 clock cycles was defined for this testbench.

The code is represented in the code A.9.

```
//Generates clock
initial begin
        #20;
        forever #20 clk = !clk;
end

//Stops testbench after 30 clock cyles
always@(posedge clk)
begin
        counter_finish = counter_finish + 1;

        if(counter_finish == 30) $finish;
end
```

Code A.9: Clock generation for the normal testbench

The behavior of the driver follows in the code A.10.

```
//Driver
always@(posedge clk)
begin
        //State 0: Drives the signal en_o
        if(counter_drv==0)
        begin
                en_i = 1'b1;
                state_drv = 1;
        end

        if(counter_drv==1)
        begin
                en_i = 1'b0;
        end

        case(state_drv)
                //State 1: Transmits the two inputs ina and inb
```

```
            1: begin
                    ina = tx_ina[1];
                    inb = tx_inb[1];

                    tx_ina = tx_ina << 1;
                    tx_inb = tx_inb << 1;

                    counter_drv = counter_drv + 1;
                    if(counter_drv==2) state_drv = 2;
            end

            //State 2: Waits for the DUT to respond
            2: begin
                    ina = 1'b0;
                    inb = 1'b0;
                    counter_drv = counter_drv + 1;

                    //After the supposed response, the TB starts over
                    if(counter_drv==6)
                    begin
                            counter_drv = 0;
                            state_drv = 0;

                            //Restores the values of ina and inb
                            //to send again to the DUT
                            tx_ina <= 2'b11;
                            tx_inb = 2'b10;
                    end
            end
    endcase
end
```

Code A.10: Part of the driver

For this testbench, we are sending the values of *tx_ina* and *tx_inb* to the DUT, they are defined in the beginning of the testbench (you can see the complete code attached to this guide).

We are sending the same value multiple times to see how the driver behaves by sending consecutive transactions.

After the execution of the Makefile, a file named *simpleadder.dump* will be created by VCS. To see the waveforms of the simulation, we just need to open it with DVE.

The waveform for the driver is represented in the figure A.9.

It's possible to see that the driver is working as expected: it drives the signal *en_i* on and off as well the DUT inputs *ina* and *inb* and it waits for a response of the DUT before sending the transaction again.

Figure A.9: Driver waveform

A.6.2 Implementing the UVM driver

After we have verified that our driver behaves as expected, we are ready to move the code into the run phase as seen in the code A.11.

```
virtual task drive();
        simpleadder_transaction sa_tx;
        integer counter = 0, state = 0;
        vif.sig_ina = 0'b0;
        vif.sig_inb = 0'b0;
        vif.sig_en_i = 1'b0;

        forever begin
                if(counter==0) begin
                        //Gets a transaction from the sequencer and
                        //stores it in the variable 'sa_tx'
                        seq_item_port.get_next_item(sa_tx);
                end

                @(posedge vif.sig_clock)
                begin
                        if(counter==0) begin
                                vif.sig_en_i = 1'b1;
                                state = 1;
                        end

                        if(counter==1) begin
                                vif.sig_en_i = 1'b0;
                        end

                        case(state)
                                1: begin
                                        vif.sig_ina = sa_tx.ina[1];
                                        vif.sig_inb = sa_tx.inb[1];

                                        sa_tx.ina = sa_tx.ina << 1;
```

```
                    sa_tx.inb = sa_tx.inb << 1;

                    counter = counter + 1;
                    if(counter==2) state = 2;
                end

                2: begin
                    vif.sig_ina = 1'b0;
                    vif.sig_inb = 1'b0;
                    counter = counter + 1;

                    if(counter==6) begin
                    counter = 0;
                    state = 0;

                    //Informs the sequencer that the
                    //current operation with
                    //the transaction was finished
                    seq_item_port.item_done();
                    end
                end
            endcase
        end
    end
endtask: drive
```

Code A.11: Task for the *run_phase()*

The ports of the DUT are acessed through the virtual interface with *vif.* $<$*signal*$>$ as can be seen in lines 4 to 6.

Lines 12 and 50 use a special variable from UVM, the *seq_item_port* to communicate with the sequencer. The driver calls the method *get_next_item()* to get a new transaction and once the operation is finished with the current transaction, it calls the method *item_done()*. If the driver calls *get_next_item()* but the sequencer doesn't have any transactions left to transmit, the current task returns.

This variable is actually a UVM port and it connects to the export from the sequencer named *seq_item_export*. The connection is made by an upper class, in our case, the agent. Ports and exports are going to be further explained in chapter 7.0.10.

This concludes our driver, the full code for the driver can be found in the file *simpleadder_driver.sv*. In the figure A.10, the state of the verification environment with the driver can be seen.

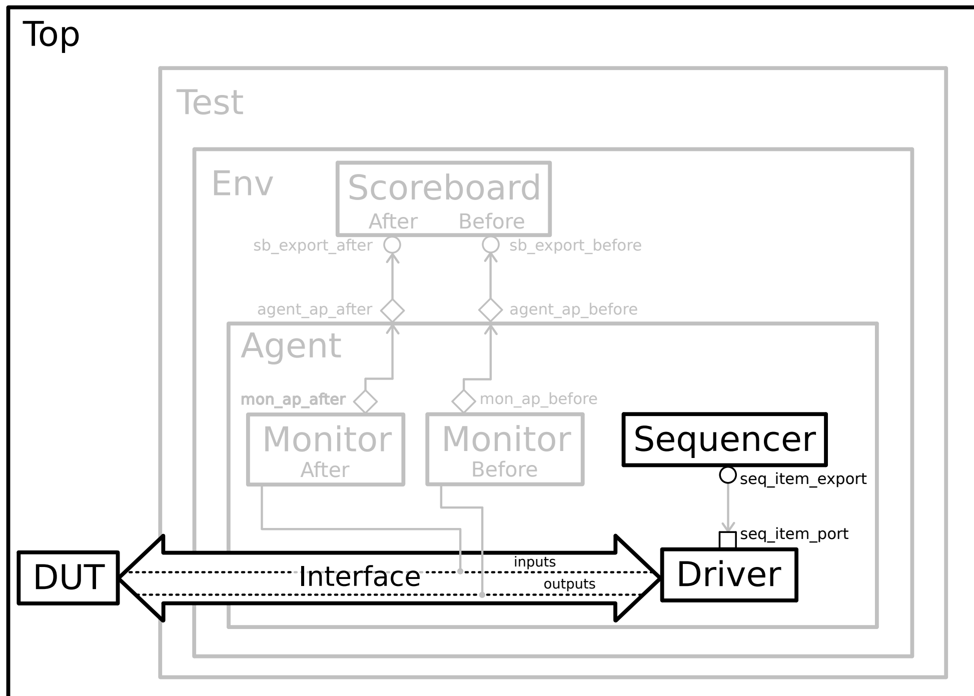

Figure A.10: State of the verification environment with the driver

A.7 Monitor

The monitor is a self-contained model that observes the communication of the DUT with the testbench. At most, it should observe the outputs of the design and, in case of not respecting the protocol's rules, the monitor must return an error.

The monitor is a passive component, it doesn't drive any signals into the DUT, its purpose is to extract signal information and translate it into meaningful information to be evaluated by other components. A verification environment isn't limited to just one monitor, it can have multiple of them.

The monitors should cover:

- The outputs of the DUT for protocol adherence
- The inputs of the DUT for functional coverage analysis

The approach we are going to follow for this verification plan is: sample both inputs, make a prediction of the expected result and compare it with the result of the DUT.

Consequently, we are going to create two different monitors:

- The first monitor, *monitor_before*, will look solely for the output of the device and it will pass the result to the scoreboard.
- The second monitor, *monitor_after*, will get both inputs and make a prediction of the expected result. The scoreboard will get this predicted result as well and make a comparison between the two values.

A portion of the code for both monitors can be seen in the code A.12 and in the code A.13.

```
class simpleadder_monitor_before extends uvm_monitor;
	`uvm_component_utils(simpleadder_monitor_before)

	uvm_analysis_port#(simpleadder_transaction) mon_ap_before;

	virtual simpleadder_if vif;

	function new(string name, uvm_component parent);
		super.new(name, parent);
	endfunction: new

	function void build_phase(uvm_phase phase);
		super.build_phase(phase);

		void'(uvm_resource_db#(virtual simpleadder_if)::read_by_name
			(.scope("ifs"), .name("simpleadder_if"), .val(vif)));
		mon_ap_before = new(.name("mon_ap_before"), .parent(this));
	endfunction: build_phase

	task run_phase(uvm_phase phase);
		//Our code here
	endtask: run_phase
endclass: simpleadder_monitor_before
```

Code A.12: Code for *monitor_before*

```
class simpleadder_monitor_after extends uvm_monitor;
	`uvm_component_utils(simpleadder_monitor_after)

	uvm_analysis_port#(simpleadder_transaction) mon_ap_after;

	virtual simpleadder_if vif;

	simpleadder_transaction sa_tx_cg;

	covergroup simpleadder_cg;
		ina_cp:		coverpoint sa_tx_cg.ina;
		inb_cp:		coverpoint sa_tx_cg.inb;
		cross ina_cp, inb_cp;
	endgroup: simpleadder_cg

	function new(string name, uvm_component parent);
		super.new(name, parent);
		simpleadder_cg = new;
	endfunction: new

	function void build_phase(uvm_phase phase);
```

```
            super.build_phase(phase);

            void'(uvm_resource_db#(virtual simpleadder_if)::read_by_name
                    (.scope("ifs"), .name("simpleadder_if"), .val(vif)));
            mon_ap_after= new(.name("mon_ap_after"), .parent(this));
        endfunction: build_phase

        task run_phase(uvm_phase phase);
            //Our code here
        endtask: run_phase
endclass: simpleadder_monitor_after
```

Code A.13: Code for *monitor_after*

The skeleton of both monitors is very similar to the driver, except for Lines 4. They represent one of the existing UVM communication ports. These ports allow different objects to pass transactions between them. In the section 7.0.10 you can consult a brief explanation of UVM ports.

The monitors will collect transactions from the virtual interface and use the analysis ports to send those transactions to the scoreboard. The code for the run phase can be designed the same way as for the driver but it was omitted in this section.

The full code for both monitors can be found in the file *simpleadder_monitor.sv*.

The the state of our verification environment after the monitors can be consulted in the figure A.11.

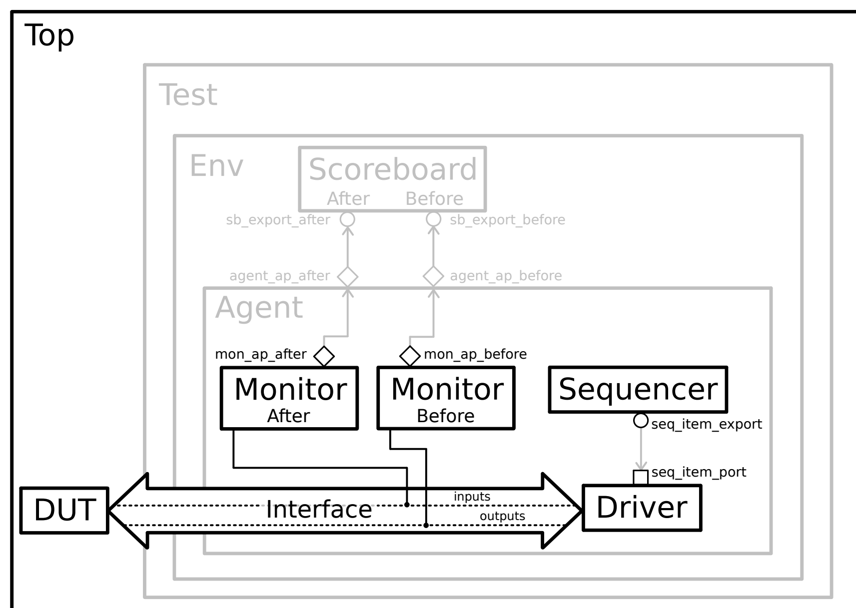

Figure A.11: State of the verification environment after the monitors

A.7.1 TLM ports

In chapter 5, it was mentioned that transactions are the most basic data transfer in a verification environment but another question arises: how do transactions are moved between components? We have already talked about TLM before when we were designing the driver. The way the driver gets transactions from the sequencer, it's the same way the scoreboard gets them from the monitors: through TLM.

TLM stands for *Transaction Level Modeling* and it's a high-level approach to modeling communication between digital systems. This approach is represented by two main aspects: *ports* and *exports*.

A TLM port defines a set of methods and functions to be used for a particular connection, while an export supplies the implementation of those methods. Ports and exports use transaction objects as arguments.

We can see a representation of a TLM connection in the figure A.12.

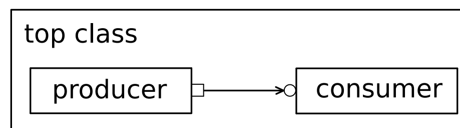

Figure A.12: Port-export communication

The communication is very easy to understand. The consumer implements a function that accepts a transaction as an argument and the producer calls that very function while passing the expected transaction as argument. The top block connects the producer to the consumer.

A sample code is provided in the table A.1.

The class *topclass* connects the producer's *test_port* to the consumer's *test_export* using the *connect()* method. Then, the producer executes the consumer's function *testfunc()* through *test_port*.

A particular characteristic of this kind of communication is that a port can only be connected to a single export. But there are cases when we might be interested in having a special port that can be plugged into several exports.

A third type of TLM port exists to cover these kind of cases: the analysis port.

An analysis port works exactly like a normal port but it can detect the number of exports that are connected to it and every time a required function is asked through this port, all other components whose exports are connected to an analysis port are going to be triggered.

class topclass extends uvm_component;
    ...
    function void connect_phase(uvm_phase phase);
        ...
        producer.test_port.connect(consumer.test_export)
        ...
    endfunction;
endclass;	

class producer extends uvm_component;
uvm_blocking_put_port#(test_transaction) test_port;
 	
    ...
    task run();
        test_transaction t;
        test_port.testfunc(t);
    endtask
endclass
	class consumer extends uvm_component;
	uvm_blocking_put_imp#(test_transaction,
	            consumer) test_export;
	    ...
	    task testfunc(test_transaction t);
	        //Code for testfunc() here...
	    endtask
	
	endclass

Table A.1: Sample code for ports and exports

In the figure A.13 it's represented an analysis port communication.

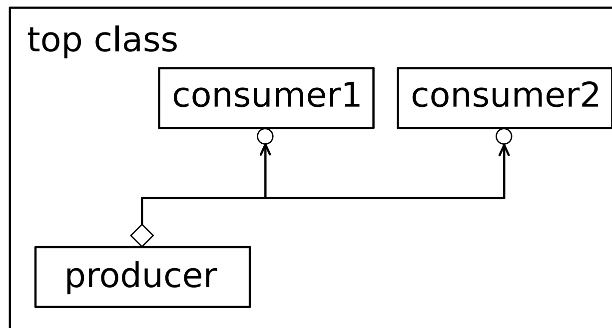

Figure A.13: Analysis port communication

The communication models mentioned here are part of Transaction Level Modeling 1.0. There is another variant, TLM 2.0, that works with sockets instead of ports, but they aren't going to be mentioned in this training guide.

A brief summary of these ports and exports can be seen in the table A.2.

Symbol	Type	Port declaration
□	Port	uvm_blocking_put_port #(transaction) port_name
○	Export	uvm_blocking_put_imp #(transaction, classname) export_name
◇	Analysis Port	uvm_analysis_port #(transaction) analysis_port_name

Table A.2: Sum up of TLM-1.0 ports

For more information about TLM, you can consult:

- Accellera's UVM 1.1 User's Guide, page 11.

A.8 Agent

We have both monitors, the sequencer and the driver, so the next step is to connect them up. This is a job for the agent.

An agent doesn't require a run phase, there is no simulation code to be executed in this block but there will be a connect phase, besides of the build phase.

We will construct the monitors, the sequencer and the driver in the build phase. We will also need to create two analysis ports, these ports will act as proxies for the monitors to be connect to an external scoreboard through the agent's ports.

Note: We could have made the connection from the monitors directly to the scoreboard within the Env class without passing by the agent's ports. There are cases when this is the best option, or other cases when it's not. It's always up to the designer to decide the best option.

After we have constructed the components we need, we have to make the connections between them. Using the concepts learned in chapter 7.0.10 about TLM ports, we can connect each port to its destination.

A part of the code for the agent can be seen in the code A.14.

```
class simpleadder_agent extends uvm_agent;
        `uvm_component_utils(simpleadder_agent)

        //Analysis ports to connect the monitors to the scoreboard
        uvm_analysis_port#(simpleadder_transaction) agent_ap_before;
        uvm_analysis_port#(simpleadder_transaction) agent_ap_after;

        simpleadder_sequencer           sa_seqr;
        simpleadder_driver              sa_drvr;
        simpleadder_monitor_before      sa_mon_before;
        simpleadder_monitor_after       sa_mon_after;

        function new(string name, uvm_component parent);
                super.new(name, parent);
        endfunction: new

        function void build_phase(uvm_phase phase);
                super.build_phase(phase);

                agent_ap_before = new(.name("agent_ap_before"), .parent(this));
                agent_ap_after  = new(.name("agent_ap_after"), .parent(this));
```

```
        sa_seqr          = simpleadder_sequencer::type_id::create(...
        sa_drvr          = simpleadder_driver::type_id::create(...
        sa_mon_before    = simpleadder_monitor_before::type_id::create(...
        sa_mon_after     = simpleadder_monitor_after::type_id::create(...
    endfunction: build_phase

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        sa_drvr.seq_item_port.connect(sa_seqr.seq_item_export);
        sa_mon_before.mon_ap_before.connect(agent_ap_before);
        sa_mon_after.mon_ap_after.connect(agent_ap_after);
    endfunction: connect_phase
endclass: simpleadder_agent
```

Code A.14: Code for the agent

In the figure A.14, it's represented the current state of our testbench.

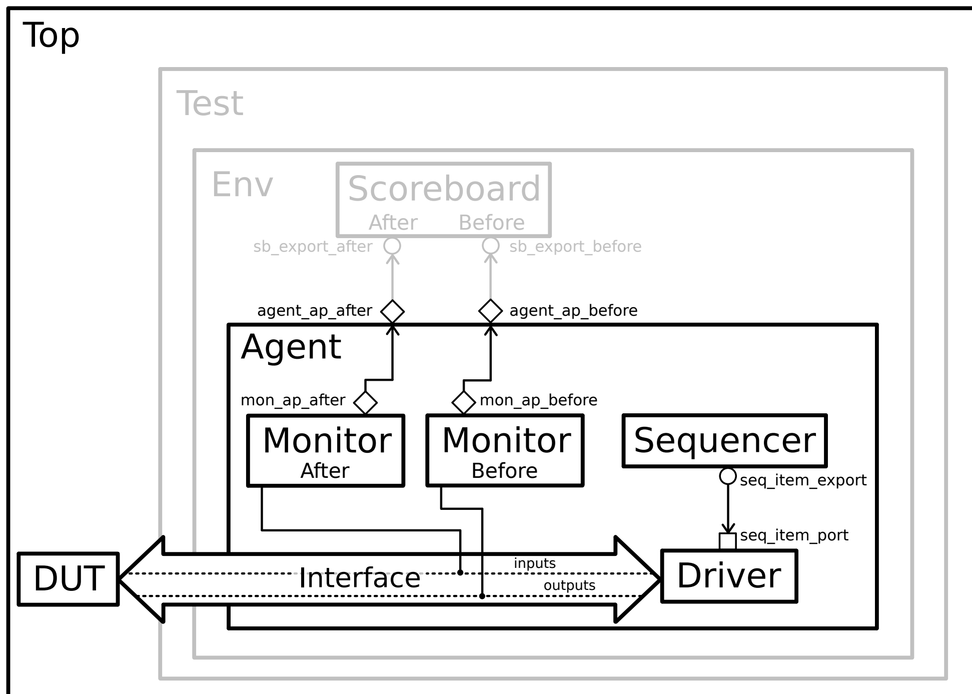

Figure A.14: State of the testbench after the agent

For more information about agents you can consult:

- Accellera's UVM 1.1 User's Guide, page 43
- Verification Academy's UVM Cookbook, page 42

A.9 Scoreboard

The scoreboard is a crucial element in a self-checking environment, it verifies the proper operation of a design at a functional level. This component is the most difficult one to write, it varies from project to project and from designer to designer.

In our case, we decided to make the prediction of the DUT functionality in the monitors and let the scoreboard compare the prediction with the DUT's response. But there are designers who prefer to leave the prediction to the scoreboard. So the functionality of the scoreboard is very subjective.

In the agent, we created two monitors, as a result, we will have to create two analysis exports in the scoreboard that are going to be used to retrieve transactions from both monitors. After that, a method *compare()* is going to be executed in the run phase and compare both transactions. If they match, it means that the testbench and the DUT both agree in the functionality and it will return an "OK" message.

But we have a problem: we have two transaction streams coming from two monitors and we need to make sure they are synchronized. This could be done manually by writing appropriated *write()* functions but there is an easier and cleaner way of doing this: by using UVM FIFO.

These FIFO will work as it's represented in the figure A.15.

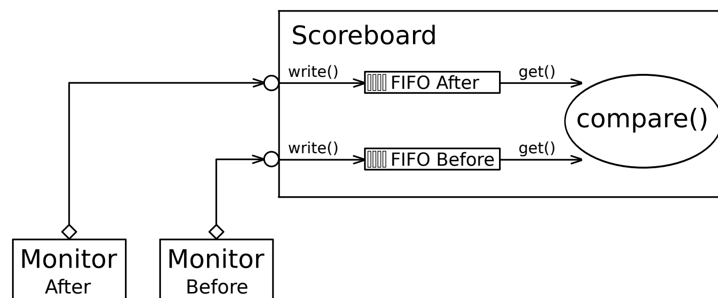

Figure A.15: Usage of FIFO in the scoreboard

The FIFO are instantiated similarly to ports/exports, with *uvm_tlm_analysis_fifo #(generic_transaction) generic_fifo* and they already implement the respective *write()* functions that are called from the monitors. To access their data we just execute the *get()* method from each FIFO.

The code from the scoreboard follows in the code A.15.

```
class simpleadder_scoreboard extends uvm_scoreboard;
	`uvm_component_utils(simpleadder_scoreboard)
```

```
	uvm_analysis_export #(simpleadder_transaction) sb_export_before;
	uvm_analysis_export #(simpleadder_transaction) sb_export_after;

	uvm_tlm_analysis_fifo #(simpleadder_transaction) before_fifo;
	uvm_tlm_analysis_fifo #(simpleadder_transaction) after_fifo;

	simpleadder_transaction transaction_before;
	simpleadder_transaction transaction_after;

	function new(string name, uvm_component parent);
		super.new(name, parent);
		transaction_before	= new("transaction_before");
		transaction_after	= new("transaction_after");
	endfunction: new

	function void build_phase(uvm_phase phase);
		super.build_phase(phase);
		sb_export_before	= new("sb_export_before", this);
		sb_export_after		= new("sb_export_after", this);

		before_fifo		= new("before_fifo", this);
		after_fifo		= new("after_fifo", this);
	endfunction: build_phase

	function void connect_phase(uvm_phase phase);
		sb_export_before.connect(before_fifo.analysis_export);
		sb_export_after.connect(after_fifo.analysis_export);
	endfunction: connect_phase

	task run();
		forever begin
			before_fifo.get(transaction_before);
			after_fifo.get(transaction_after);
			compare();
		end
	endtask: run

	virtual function void compare();
		if(transaction_before.out == transaction_after.out) begin
			`uvm_info("compare", {"Test: OK!"}, UVM_LOW);
		end else begin
			`uvm_info("compare", {"Test: Fail!"}, UVM_LOW);
		end
	endfunction: compare
endclass: simpleadder_scoreboard
```

Code A.15: Code for the scoreboard

In the figure A.16, it's represented the current state of our testbench.

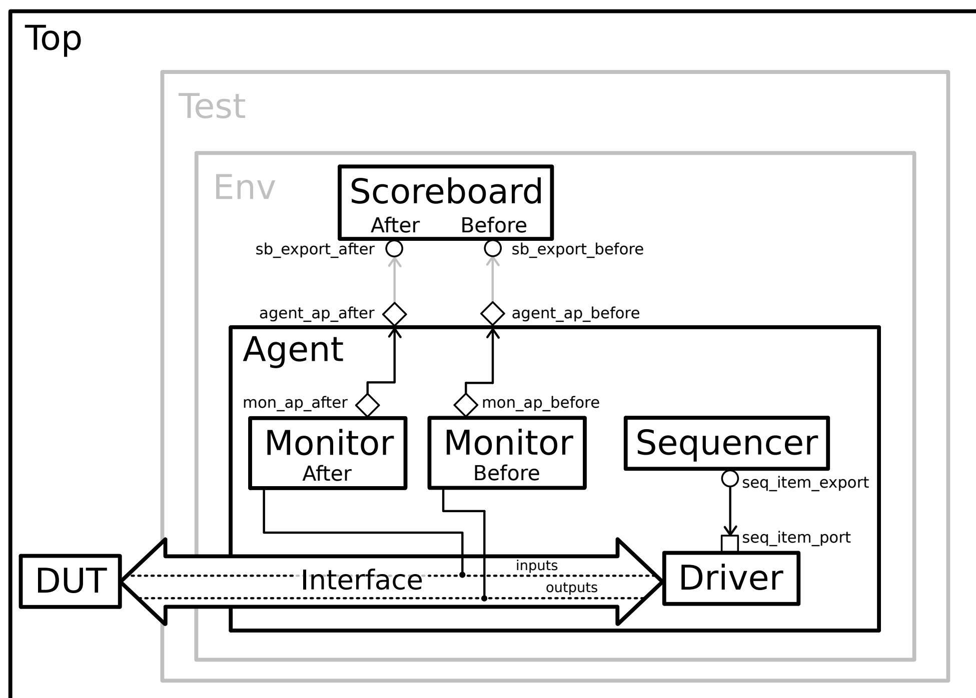

Figure A.16: State of the testbench after the scoreboard

For more information about scoreboards you can consult:

- Accellera's UVM 1.1 User's Guide, page 72
- Verification Academy's UVM Cookbook, pages 155 and 163
- Comprehensive Functional Verification: The Complete Industry Cycle, J. Goss, page 82

A.10 Env

We are getting close to have a working testbench, there are two classes missing: the *env* and the *test*.

The *env* is a very simple class that instantiates the agent and the scoreboard and connects them together.

The source is represented in the code A.16.

```
class simpleadder_env extends uvm_env;
	`uvm_component_utils(simpleadder_env)

	simpleadder_agent sa_agent;
	simpleadder_scoreboard sa_sb;

	function new(string name, uvm_component parent);
		super.new(name, parent);
	endfunction: new

	function void build_phase(uvm_phase phase);
		super.build_phase(phase);
		sa_agent	= simpleadder_agent::type_id::create(...
		sa_sb		= simpleadder_scoreboard::type_id::create(...
	endfunction: build_phase

	function void connect_phase(uvm_phase phase);
		super.connect_phase(phase);
		sa_agent.agent_ap_before.connect(sa_sb.sb_export_before);
		sa_agent.agent_ap_after.connect(sa_sb.sb_export_after);
	endfunction: connect_phase
endclass: simpleadder_env
```

Code A.16: Code for the *env*

In the figure A.17, it's represented the current state of our testbench. There is only one component left now: the *test* class.

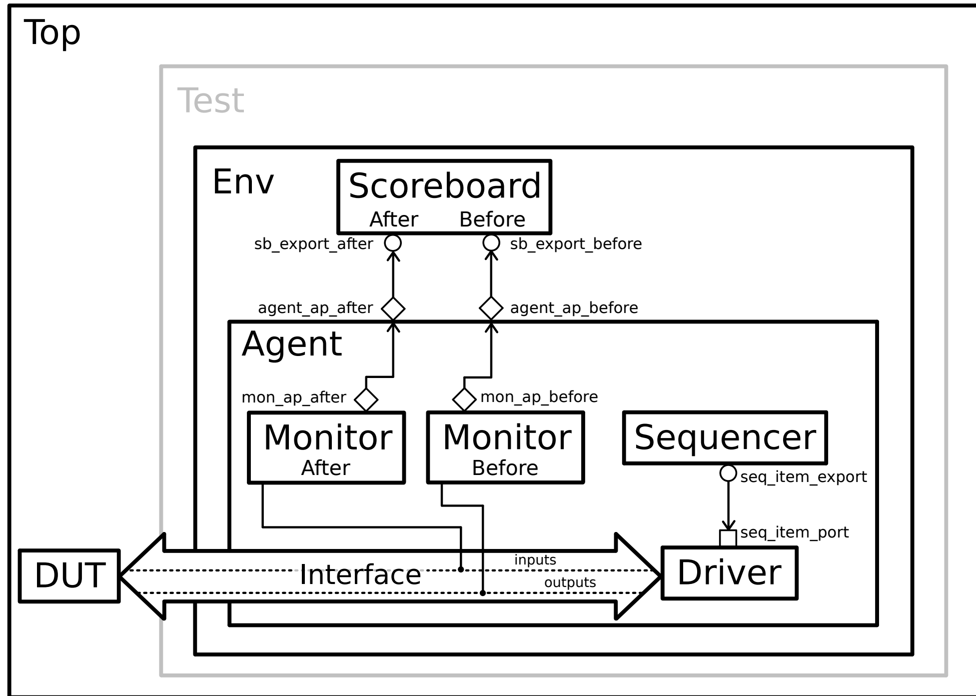

Figure A.17: State of the testbench after the env

A.11 Test

At last, we need to create one more block: the *test*. This block will derive from the *uvm_test* class and it will have two purposes:

- Create the *env* block
- Connect the sequencer to the sequence

You might be wondering why are we connecting the sequencer and the sequence in this block, instead of the agent block or the sequence block. The reason is very simple: by specifying in the test class which sequence will be going to be generated in the sequencer, we can easily change the kind of data is transmitted to the DUT without messing with the agent's or sequence's code.

```
class simpleadder_test extends uvm_test;
        `uvm_component_utils(simpleadder_test)

        simpleadder_env sa_env;

        function new(string name, uvm_component parent);
                super.new(name, parent);
        endfunction: new

        function void build_phase(uvm_phase phase);
                super.build_phase(phase);
                sa_env = simpleadder_env::type_id::create(...
        endfunction: build_phase

        task run_phase(uvm_phase phase);
                simpleadder_sequence sa_seq;
                phase.raise_objection(.obj(this));
                        sa_seq = simpleadder_sequence::type_id::create(...
                        assert(sa_seq.randomize());
                        sa_seq.start(sa_env.sa_agent.sa_seqr);
                phase.drop_objection(.obj(this));
        endtask: run_phase
endclass: simpleadder_test
```

Code A.17: Code for the test class

Line 21 starts a sequencer in the desired sequence for this test.

In the figure A.18, it's represented the current state of our testbench.

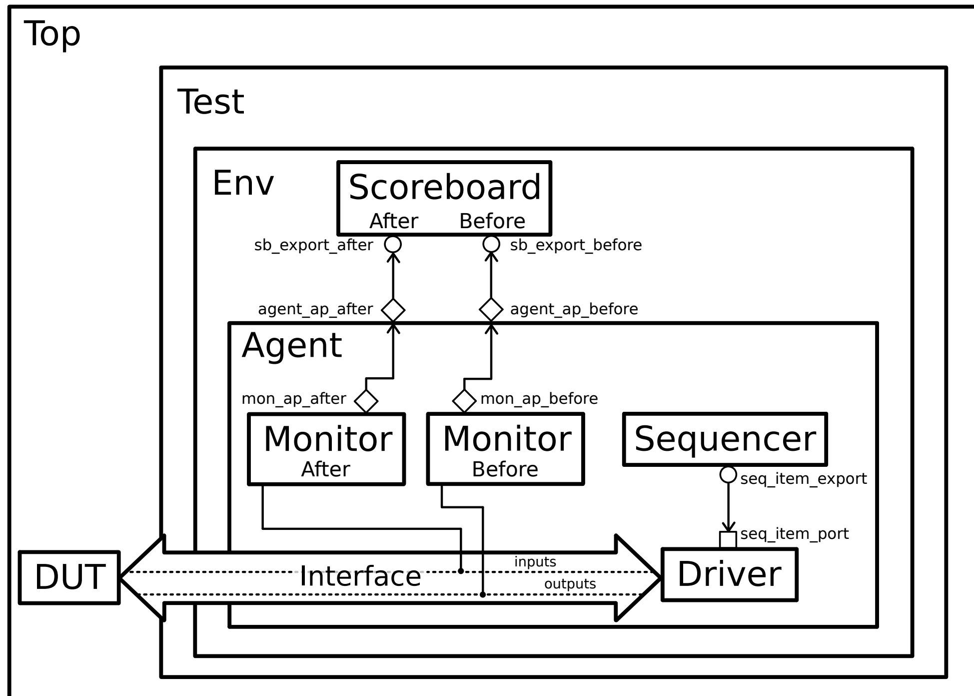

Figure A.18: Final state of the testbench

Our testbench is finally ready, now it's time to execute it and check the results.

A.12 Running the simulation

To run the simulation, we simply execute the provided Makefile:

$ make -f Makefile.vcs

The testbench will generate random inputs and then those inputs will be sent to the DUT. The monitors will capture the data in the communication bus and make a prediction of the expected result. Finally the scoreboard will evaluate the functionality by matching the DUT's response with the prediction made by one of the monitors. If the DUT and the prediction match, an "OK" message will be outputted, otherwise, we will se a "Fail" message.

So, In the output of the simulation, we must find for the messages starting with *UVM_INFO* because the *compare()* method from the scoreboard is going to print a message using the macro *'uvm_info()* with the result of the test.

The result of the simulation can be seen in the code A.18.

```
***********     IMPORTANT RELEASE NOTES     ************

You are using a version of the UVM library that has been compiled
with `UVM_NO_DEPRECATED undefined.
See http://www.eda.org/svdb/view.php?id=3313 for more details.

(Specify +UVM_NO_RELNOTES to turn off this notice)

UVM_INFO @ 0: reporter [RNTST] Running test simpleadder_test...
UVM_INFO simpleadder_scoreboard.sv(49) @ 70: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 130: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 190: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 250: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 310: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 370: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 430: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 490: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 550: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 610: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 670: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 730: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 790: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO simpleadder_scoreboard.sv(49) @ 850: uvm_test_top.sa_env.sa_sb [compare] Test: OK!
UVM_INFO ../uvm-src/uvm-1.1d/src/base/uvm_objection.svh(1268) @ 900: reporter [TEST_DONE]

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    16
```

```
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]    1
[TEST_DONE]    1
[compare]    14
$finish called from file "../uvm-src/uvm-1.1d/src/base/uvm_root.svh", line 430.
$finish at simulation time                900
VCS Simulation Report
Time: 900 ns
CPU Time:      2.040 seconds;      Data structure size:   0.2Mb
```

Code A.18: Result of the simulation

Bibliography

[1] Verification Academy. *UVM Cookbook*. Mentor Graphics, 2013.

[2] Accellera. Universal Verification Methodology (UVM) 1.1 Class Reference. 2011.

[3] Accellera. *Universal Verification Methodology (UVM) 1.1 User's Guide*. Accellera, 2011.

[4] Janick Bergeron. *Writing testbenches: functional verification of HDL models*. Kluwer Academic Publishers, 2003.

[5] Stephen A. Edwards. Design and Verification languages. *Columbia University Computer Science Technical Reports*, (CUCS-046-04), 2004.

[6] Mark Glasser and Tom Fitzpatrick. *Advanced Verification Methodology Cookbook*. 2008.

[7] Nisvet Jusic and Jan Nillson. Design and Verification languages. 2007.

[8] GE Moore. Cramming more components onto integrated circuits. 86(1):82–85, 1965.

[9] S Rosenberg and KA Meade. *A practical guide to adopting the universal verification methodology (UVM)*. lulu.com, 2010.

[10] Louis Scheffer, Luciano Lavagno, and Grant Martin. *EDA for IC system design, verification, and testing*. CRC Press, 2006.

[11] National Semiconductor. LM4550 AC'97 Rev 2.1 Multi-Channel Audio Codec Datasheet, 2003.

[12] NXP Semiconductors. UM10204 I2C-bus specification and user manual. *Rev*, (April), 2007.

[13] Chris Spears. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2007.

[14] Verisity. *e Reuse Methodology (eRM) Developer Manual*. 2004.