POLITECNICO DI TORINO

Master's Degree in Electronic Engineering
Embedded Systems

Politecnico
di Torino

Master's Degree Thesis

UVM testbench development for I2S protocol verification

Supervisors

Prof. MAURIZIO MARTINA

Eng. ALESSIO PELLE

Eng. SANDRO SARTONI

Candidate

DAVIDE FERRARO

October 2024

Abstract

Modern digital integrated circuits are increasingly becoming more and more complex, with several internal logic blocks, submodules and interconnections. Not only the functionalities they implement are quite complicated, but they must perform their tasks with stringent timing, area and power requirements.
Due to these reasons, verification of integrated circuits is becoming more and more important. Thanks to the verification process, it is possible to make sure that new devices behave according to their specifications in a timely manner, reducing the time to market. Moreover, it allows to significantly shrink the number of bugs reaching the physical production stage of chips, preventing costly manufacturing errors and increasing the product reliability.

The introduction of the Universal Verification Method (UVM) has drastically changed the way verification is carried out, introducing a standardized, robust, modular and flexible environment, enhancing the interoperability and facilitating the exchange of code between different teams and projects.
This thesis aims at verifying an I2S protocol master unit developed by TDK Invensense, using the SystemVerilog language and the UVM framework.
After a preliminary study on UVM and SystemVerilog language through Cadence courses, the obtained notions have been applied to develop a full UVM environment with classes and tests, written to cover one, or more, specific features of the Device Under Test (DUT), needed to fully verify the DUT.
The results obtained by applying the aforementioned tests have been thoroughly analyzed and evaluated. Neat and intuitive coverage reports have been generated through the use of regression testing and merging in combination with SystemVerilog and UVM coverage features.
The generated reports showed a quite complete verification of all specifications written in the Test Plan. The remaining not covered points have been investigated to understand whether they represent not reachable scenarios or tests related bugs. For the few reachable not covered points, some ad hoc tests have been created by driving specific signals to recreate the correct stimulus generation, while the others were removed through refinement steps adding justification on why could not be cover.
Thanks to the combination of the random stimuli generation and the constrained one, a full coverage of the master unit has been reached.

Table of Contents

List of Figures

Listings

Acronyms

ABV
Assertion Based Verification

DUT
Device Under Test

eRM
e-Reuse Methodology

FIFO
First In First Out

FSM
Finite State Machine

I2S
Inter-IC Sound

LSB
Least Significant Bit

MDV
Metric Driven Verification

MEMS
Micro-Electromechanical Systems

MSB
Most Significant Bit

OVM

Open Verification Methodology

RTL

Register-Transfer Level

RX

Reception

SCK

Continuous Serial Clock

SD

Serial Data

TLM

Transaction Level Modeling

TX

Transmission

UVC

UVM Verification Component

UVM

Universal Verification Methodology

VIP

Verification Intellectual Property

VMM

Verification Methodology Manual

WS

Word Select

Chapter 1

Introduction

1.1 Background

The electronic industry is in constant development and innovation, continuously pushing the boundaries to achieve greater results in terms of performance, efficiency and features.
The evolution of the systems has led to an exponential increase of the functionalities offered. This growth has been accompanied by a correspondent rise in the complexity required to meet all the expected specifications.
The logical consequence of these advancements has been an exponential technology scaling, which corresponds to the rapid increase of the units number present on a single chip or system. However this innovation introduced new challenges and issues that have to be checked and controlled such as the power management that has to be optimized maintaining top-notch performance without compromising energy efficiency, or the coexistence of complex analog and digital units with their communication problems.

Due to the parallel compelling need to be competitive in the market both on timing and performances ends, the electronic industry continuously tries to create methodologies to reduce time and costs required to create a new product. Design and verification are certainly the two most expensive process in term of time, and the latter is the one that usually is the most time consuming and thus the most critical in the overall development process.

Verification is a fundamental production stage: to verify means to check that the units, circuits and systems comply with the required specifications according to different parameters as well as performance, power and timing. The relevance of each of the aforementioned characteristics depends on the nature of the system to

be tested: in an embedded system power management is a very delicate topic while in real time systems it is compulsory that each data is not only logically correct but also that it is computed in the correct time frame.
Verification has been proven over time as a fundamental step to avoid wasting resources and incurring in unnecessary costs. As a matter of facts, there have been cases where companies have been forced to remove one of their products off the market due to unexpected behaviours or malfunctioning. Such cases lead not only to a non negligible money loss, but also to a dent in the company reputation. Those issues could have been avoided by performing a tailored verification routine starting from the digital design stage.
Digital verification allows to identify issues before the circuit is physically printed, preventing huge amounts of losses in terms of money and time. As a consequence, this topic has been thoroughly studied and applied both in academia and in industry. Through the use of an ad-hoc environment, the digital circuit is tested by means of logic simulations that perform the following fundamental tasks:

- the injection of a pseudo-randomic set of input stimuli to cover many different scenarios;
- the collection of output data gathered from the application of the aforementioned stimuli;
- generation of a reference model used to check data correctness;
- comparison of collected data to the expected one, thus making sure the DUT works as intended;

The necessity of reducing verification time has led to the creation of programming languages designed to enhance the precision and to ease overall the whole verification process by adding specific features to obtain better results.
With the introduction of new methodologies, the industry continues to improve its capabilities of generating reliable high quality products reducing at the minimum the probability of unexpected and incorrect behaviour.

1.2 UVM and MDV

Due to the aforementioned reasons, verification plays a central role in the production line of today's most advanced electronic systems.
As the electronic systems have evolved over time verification methodologies progressed as well, thus leading to a significant progress since the early 2000s.

In 2001-2002 the e-Reuse Methodology (eRM) first emerged. This was one of the first attempts at introducing a standardized approach to verification — including

guidelines — such as the functional partitioning of the testbench and a file naming convention [1].

Later on, in 2005, Synopsys announced the release of the Verification Methodology Manual (VMM), a manual that established guidelines and recommendations for object-oriented programming by taking advantage of randomization and constraints which are essential in verification. The VMM also provided industry best practises to help prevent common mistakes creating new verification components. [2]
After that, it is important to highlight another key stone: in 2008 the Open Verification Methodology (OVM) was created, becoming the first truly open, interoperable and proven verification methodology. OVM is a SystemVerilog language class library, completely open-source that defines a framework for reusable Verification IP (VIP). That has been one of the key moments for the creation of a universal standard. Indeed, one year later the so called Open Verification Methodology has been chosen by Accelera to become the base of the nowadays used Universal Verification Methodology (UVM) [3].
Using OVM as its core, UVM integrates years of object-oriented designs and methodologies in order to create scalable, reusable, and flexible testbenches. These principles have been the cornerstones of UVM, a methodology with the main goal of creating an open, unified class structure for interoperable VIP enhancing code reusability and adding strong built-in automation features [4, 5].

Another significant advantage of UVM is time optimization in industry projects. Thanks to the object-oriented approach and common generic guidelines, it is possible to separate different parts of a testbench between various team members enhancing productivity and facilitating the reuse of the single developed components. Splitting work between different people has had a large impact on the testbench produced, obtaining a more modular and reusable verification environment connected to the Device Under Test (DUT).
By following the UVM approach and taking advantage of the randomization and coverage-oriented characteristics of the SystemVerilog language, it is possible to create powerful testbenches capable of verifying the majority of, if not all, the possible inputs and scenarios that devices to be verified can experiment and undergo through during their operative lifetime.

Moreover another layer of methodology is often used: the Metric Driven Methodology (MDV). The MDV is used to defines clear and measurable goals of the verification creating Executable Plans of what has to be tested.

Furthermore MDV offers guidelines and tools to enhance the efficiency and effectiveness of the verification process using metrics and automation to maximize the verification potential of a testbench. The main features of this methodology

can help to:

- manage frequent changes in specifications and project plans;
- generating organized and intuitive reports useful to categorize the verification results leading to better productivity and higher work quality;
- finding more bugs faster through the combination of formal techniques and constrained-random inputs;

The MDV process is divided into four stages: [6]

- **Plan**: highlights the needed specifications, structure the verification plan and design the verification environment defining the structure;
- **Construct**: the verification environment is created through the reuse of existing modules or the implementation of new ones in order to cover the verification plan scenarios;
- **Execute**: simulation is launched and a formal analysis of the result is performed. Often powerful environments are used to summarize the results obtained;
- **Measure/Analyze**: features like coverage and assertions are used to determine the effectiveness of the test performed. In case of not satisfying results, bugs or failures adjustments and refinements are performed and then the cycle is restarted;

1.3 Aim and motivation

The present thesis aims primarily at demonstrating the effectiveness of the combination between the Universal Verification Methodology (UVM) and the Metric Driven Verification (MDV) approaches. The final achievement is to verify the correctness of a Design Under Test (DUT) provided as a grey box exploiting the main and most useful characteristics of the two aforementioned methodologies. To reach a full verification of the DUT, scripting and test regressions have been extensively used, as well as SystemVerilog language features such as assertions and coverage.
The DUT analyzed is an I2S protocol master unit developed in Verilog by TDK Invensense.
As aforementioned, a grey box approach has been used bringing only a partial knowledge of the unit's design while having a thorough understanding of how the

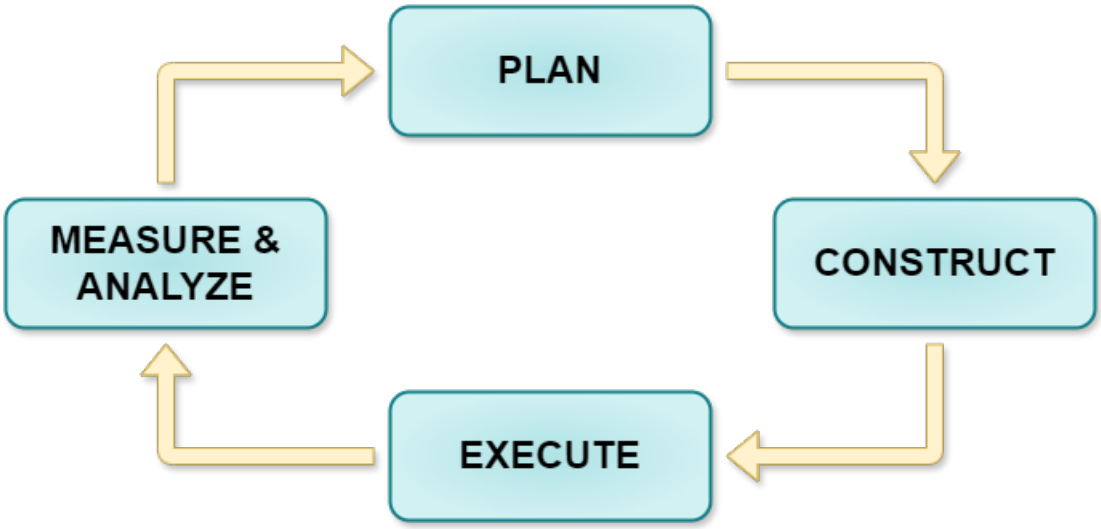

Figure 1.1: MDV Cycle

protocol should work and its expected functionalities. This strategy allows to create a tailored testbench environment without being influenced by the design choices.

Prior to the beginning of the work on the practical aspects of this thesis project, I dedicated a significant amount of time to acquire and develop the necessary verification related skills and knowledge.
TDK Invensense provided invaluable support giving me the opportunity to attend Cadence courses on: SystemVerilog for Design and Verification, SystemVerilog Verification with UVM, SystemVerilog Assertions and on Cadence software such as: Xcelium Simulator, Xcelium Fault Simulator, Xcelium Integrated Coverage (which also introduced the usage of IMC a software for the coverage and reports). The combination of acquired theoretical concepts and the experience gained during months have been crucial to successfully complete the verification of the I2S protocol master unit.

Chapter 2

I2S Protocol

2.1 Introduction

The I2S protocol (Inter-IC-Sound), also known as IIS, was developed in 1986 by Philips Semiconductor (now known as NXP Semiconductor).
The I2S protocol has been designed as a standardized solution for the transmission of digital audio data between different units such as microcontrollers.
As electronics became a fundamental part of the industry and the world transitioned from analog to digital, a new protocol was needed that could transmit data without losing quality while maintaining a high efficiency.

Over the years, I2S has proven to be crucial in applications where the sound quality is critical, becoming a de-facto standard in the digital industry. Moreover, the relatively low complexity of the protocol and its low number of interface pins helped to make it one of the most convenient choices in audio design systems.
The establishment of the I2S has been essential for the rapid spread of digital audio systems in the consumer and professional markets. Before its adoption, audio systems relied on analog signals that were subject to a lot of issues, such as distortions or interferences [7].

The primary goal behind the creation of the I2S standard was to establish a simple and reliable solution for the transmission of digital audio signals between various components of a digital circuit or between different chips. This ensures an excellent sound quality and great signal robustness compared to the previous technologies.
Even nowadays, the forever long debate to determine the superiority between analog and digital audio quality hasn't concluded yet. A consequence of this issue is the production of different systems' configurations according to its final use and

its final consumer: some systems keep the audio analog as long as possible reducing to the minimum the digital processing, others, instead, take the audio data digital and process that without having any analog part.
As always in the market, all the decisions are made upon the trade off between cost and quality, so in audiophile systems there is the tendency of incorporating a larger part of analog elements prioritizing sound fidelity, while on the other hand in consumer electronics, where digital micro-electro-mechanical systems (MEMS) microphones and digital speakers are vastly present, the focus is on cost reduction using digital audio data and having simpler systems [7].

Despite this ongoing debate, the key role that the I2S has played in the digital world is undeniable and, thanks to the improvements of digital speakers and microphones, the protocol is nowadays spread in the majority of audio systems.

2.2 Specification

The I2S interface is a serial bus that uses three lines:

- Continuous Serial Clock (SCK)
- Word Select (WS)
- Serial Data (SD)

The protocol transfers data between two units called transmitter and receiver. Both transmitter and receiver units share the same clock signal for data transmission, the unit that generates the SCK and WS is referred to as the controller. In complex systems it is possible to have multiple transmitters and receivers. If not regulated this could lead to issues, as logic and electrical conflicts may arise when multiple transmitters try to broadcast a frame concurrently. In those cases the solution is to have a system controller managing the digital data flow between the units [8]. The use of only three lines is adopted as to minimize cost and area, keeping at the minimum the number of pins required and simplifying wiring.

2.2.1 Serial Data

The SD is transmitted Most Significant Bit (MSB) first in two's complement. The MSB first transmission is used in order to prevent possible word length problems. In this way, the receiver doesn't need to know how many bits the transmitter wants to send and neither the transmitter needs to know how many bits the receiver can handle. If the receiver's word length is bigger than the transmitter's word, the least significant bits that are not filled will be set to '0'. On the opposite, if the

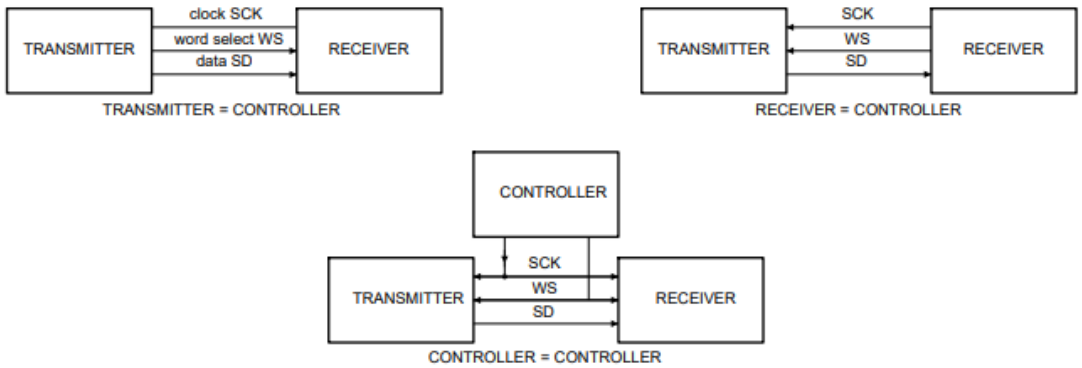

Figure 2.1: I2S Simple System Configurations [8]

transmitter's word length is greater than the receiver's one the word is truncated and bits after the LSB are ignored.
In order not to have problems between different words, the MSB of the next word is transmitted one clock period after the Word Select change, synchronizing the start of each new word on the trailing or leading clock edge [8].

2.2.2 Word Select

The WS signal specifies on which channel the transmission occurs: if WS is high, channel 0 (left) is selected, otherwise with WS low channel 1 (right) is the active one.
The WS changes one clock period before the MSB is transmitted guaranteeing synchronicity between transmitter and receiver, while giving the receiver time to store the previous word [8].

2.3 Master Block

For the purposes of this thesis a I2S Master protocol encoder block has been developed by TDK Invensense with its analysis and verification as the main target.

The internal architecture of the Master block is composed by four key components: a PROTOCOL block, a PAD block and two FIFO blocks.

In order to have a more generic code some parameters has been defined:

- **WORD_LEN_LOG2** represents the log2 of the word size transmitted or received by the block master; its value is 4 by default

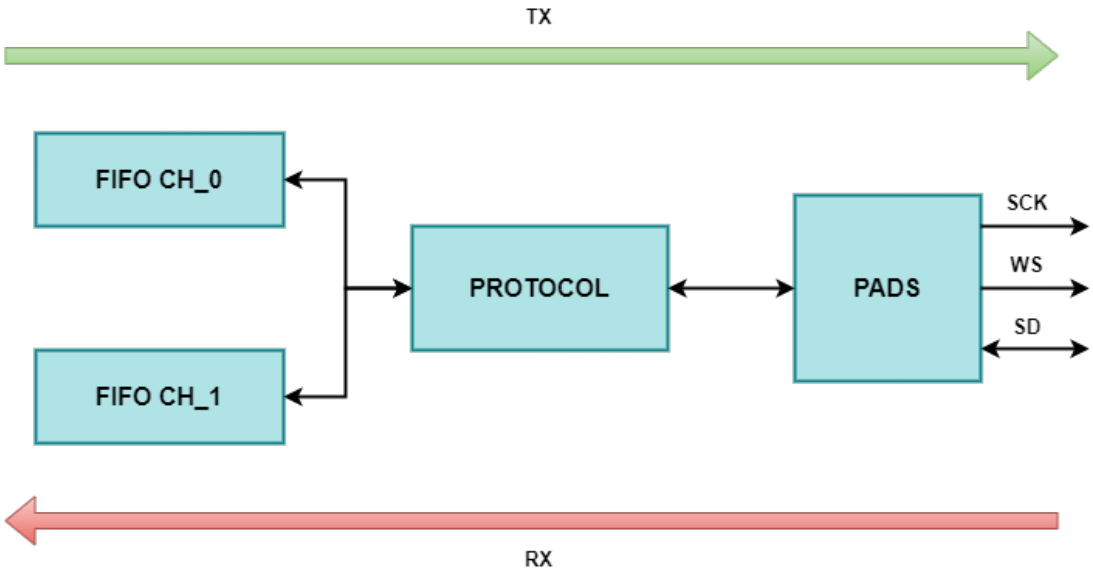

Figure 2.2: I2S Master Block Diagram

- **FIFO_ADDR_LEN_LOG2** represents the log2 of the FIFO length; its value is 3 by default
- **WORD_LEN** it is the actual word length
- **FIFO_ADDR_LEN** it is the address length

The Master block can work in two different modes: Transmission (TX) and Reception (RX).
The TX mode is responsible for transmitting data from the two channels to the output of the PAD block. In RX mode, instead, data is taken from the PAD side and transferred to the output of FIFO blocks [9].

2.3.1 FIFOs

Each FIFO block is a dual-port memory structure that can be accessed from two sides: one side is connected to the PROTOCOL block, the other to the peripheral's interface. Both FIFO sides support read and write operations for an optimal data management.
When the block is in TX mode each FIFO is written from outside of the peripheral through the interface and read by the Protocol block. In RX mode the opposite happens, with the FIFO read by the external and written by the PROTOCOL block.
The FIFOs have been implemented as circular buffers, and they are controlled

by two pointers that are updated after each read or write operation to track the presence of data saved. This approach ensures a correct data management inside FIFOs without the necessity of additional memory to store the position or addresses.
Moreover two edge-cases has to be evaluated: the possibility of having a full FIFO and the possibility of having an empty FIFO. The master block checks the happening of one of these scenario using two signals. The signal associated to FIFO full is asserted when the FIFO buffer reaches its maximum capacity (the internal pointer reaches the last possible position) preventing other write operations. Conversely, if the FIFO is empty the correspondent signal is asserted, not allowing other read operations. In case of write operation with FIFO full data will not be stored and lost, instead in case of read operation with FIFO empty the output will be '0'.

2.3.2 PROTOCOL block

The PROTOCOL block controls a Finite State Machine (FSM) that manages the control signals needed to correctly handle a transmission or a reception.
During a transmission, data is retrieved from the two FIFOs and serially transmitted to the PAD block along with the WS signal and the clock. While receiving a frame, on the other hand, the serial data is taken from the PAD, assembled into words and directed to the correct channel (FIFO) according to the WS signal.
The FSM has six possible states:

- IDLE
- STARTUP
- TX_CH0
- TX_CH1
- RX_CH0
- RX_CH1

The IDLE state is the first active state right after the device power-up and before the enable signal, cfg_i2s_en, is asserted. In this state all signals are held low.
Once the enable signal is asserted, the state transitions to STARTUP. In this state, a word of all zeros is sent to the I2S line while the clock starts toggling and WS is set high. The STARTUP state prepares and synchronizes the receiver for the word reception on the Word Select falling edge.
Following the falling edge of WS, the state transitions again from STARTUP to TX_CH0. During TX_CH0 a transition to channel 0 is performed, the first

available word is sent to the fifo associated to channel 0. After the first word has been transmitted, WS toggles and the state changes to TX_CH1 with a word moved to the FIFO of the other channel. It is important to underline that each transmission always starts on channel 0 without exceptions.
The Transmission goes on alternating the two channels (and so the correspondent states) until the enable signal transitions to 0. With this last transition, the transmission is interrupted immediately even if a word has not been fully transmitted.
In reception mode (RX) the FSM operates in a similar way, with states RX_CH0 and RX_CH1 mirroring the transmission states.
During reception, data is taken from the PAD and directed to the respective channel based on the WS signal value. Just like during the transmission, if the enable signal is deasserted the reception process is halted immediately, even though the data reception is incomplete.

2.3.3 PAD

The I2S output is managed by the PAD block.
As with the other components the behaviour of the PAD block changes according to the protocol mode.
In the PAD block there are the SD line, SCK signal and WS signal.

The SCK signal is PAD the serial clock signal. This is simply the clock signal of the protocol inverted and used to correctly synchronize the unit with the remaining of the Master block.

The WS signal mirrors the functionality of the WS in the PROTOCOL block. This signal is used to correctly select the audio channel to which data has to be directed: channel 0 if WS is 0, and channel 1 otherwise.

The SD line is the most complex one, it is a bidirectional serial line that operates according to the mode of the Master block.
In case of a transmission, the SD line works as an output by receiving data from the PROTOCOL block and transmitting it to a potential external source.
In reception mode, instead, the SD line works as an input by receiving serial data from external source and transmitting them to the PROTOCOL block.
In case of an RX, in order to prevent possible conflicts on the line, the output capability of the line can be disabled. This control ensures the correct functionality of the line in both modes, keeping the input always active and enabling the output only when necessary to transmit a specific data stream [9].

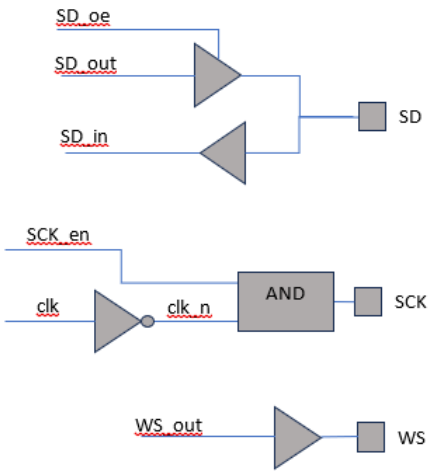

Figure 2.3: I2S Pad block [9]

Chapter 3

UVM

3.1 Key Concepts

The introduction and widespread adoption of UVM in the semiconductor industry has brought several advantages to verification that couldn't be made possible by its precursors or by the OVM approach.
UVM constructed its strenght on some key pillars [10]:

- **Reusability**: although the OVM bases its structure on a object-oriented approach with pre-defined classes, those where not designed to be modular and reusable: without those characteristics its adaptability was limited in different verification environments. Instead, UVM has been designed to set a clear separation between the Device Under Test (DUT) and the testbench. This more modular approach leads to an easier code reusability of the verification segments in different projects.
  The UVM Class Library has been one of the main advantages that allowed UVM to become so widely adopted by the industry;

- **Flexibility**: differently from OVM, the UVM approach is designed to guarantee a higher level of flexibility between the Register-Transfer Level (RTL) and the Transaction-Level Modeling (TLM), increasing the number of scenarios that can be verified;

- **Maintainability**: strictly connected to the other factors, an high standard of maintainability ensures a code that is easy to manage and update. Even after years or updates in UVM methodologies, previously written code is still functional and usable or, at least, it is very easy to be modified and corrected. This feature ensures long term efficiency of the code in different verification projects limiting the obsolescence issue;

- **Standardization**: as said, standardization is one of the strengths of UVM. While OVM introduced guidelines in verification, it lacked formal standards in how to develop code, making each piece of code written by verification engineers different and structured in a personal way. The inconsistency in the code structure brought up new problems in code sharing between teams, with lots of time wasted in trying to understand and decipher the code content and structure. With the introduction of a strict standard methodology, written code became more uniform, consistent and more manageable, ensuring an easy diffusion between different industries, too.

3.2 UVM Classes

In order to be maintainable and reusable, in the UVM Class Library all the fundamental blocks used in a testbench are derived from a set of basic classes that defines characteristics and methods that each block can have [11]:

- **uvm_void**: it is the base class of all the UVM classes.
  It has no particular functionalities and sets no restrictions,

- **uvm_object**: abstract class derived from uvm_void. All classes used to create a UVM Environment are derived from uvm_object.
  It sets some useful methods like copy, print and compare often used in the derived classes,

- **uvm_component**: derived from uvm_object, this is the direct parent class of all UVM standard components. It is used to introduce mechanisms like factory, hierarchy, reporting, objection and phasing that allow to build a real structure between all components and define how those modules interact one with the other,

- **uvm_sequence_item**: class derived from uvm_transaction that allows to manage phases and timing. It is the base class for objects, sequence items and sequences.

3.2.1 UVM Testbench Structure

From these base classes, in particular from the uvm_object, the real testbench components are declared. The most important classes that define the testbench structure are:

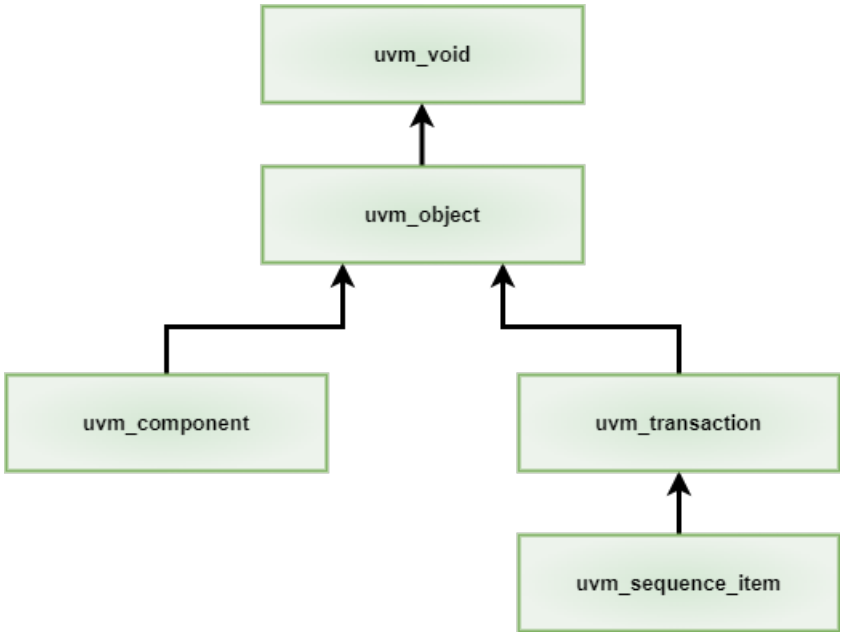

Figure 3.1: UVM hierarchy

uvm top

It is the top level module and it functions as a container where the uvm_test and the Design Under Test (DUT) will be instantiated, defining explicitly the connection between them. To guarantee the maximum code versatility and reusability, the uvm_test is usually instantiated at run_time; this dynamic instantiation allows the use of different tests according to specific settings that we need to verify. This also provides a flexible approach to verify the various DUT configurations.
uvm_top usually contains also one or more clock generation blocks, together with some control logic on the reset generation, too.
The connection of the DUT to all the testbench signals is performed through the use of a specialized object called Interface. The Interface will be further explained in detail later.

uvm test

The uvm_test has the primary aim of instantiating the environment class and configuring all the necessary parameters required. This purpose is usually achieved by using the set method and factory overrides, features that are peculiar to the UVM approach. In addition, the uvm_test is also asked to run certain sequences on a specific sequencer.
Typically, in a complex system a high number of different tests are required to

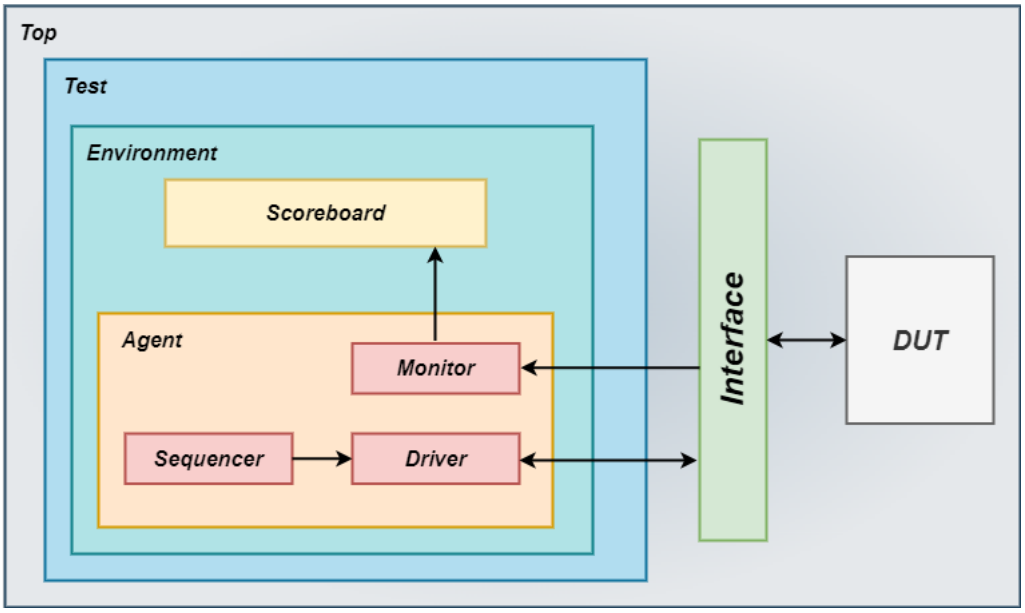

Figure 3.2: UVM Testbench Hierarchy

achieved an optimal verification. In order to obtain that, all the defined tests are derived from a common one, allowing to extend test functionalities, set values or enabling/disabling classes at need.
Each created test is used to verify one or more specific functionalities of the system under test, covering all the possible scenarios and edge cases.

uvm environment

The uvm_environment class represents another layer of encapsulation.
An environment contains a variety of reusable components useful for verification purposes. Inside an environment it is not unusual to find other layers of environment nested: this method is used for the sole purpose of enhancing reusability. With the creation of those encapsulation layer, it is easier to incorporate components, environments or agents written by different teams or developed for other projects that can be useful for the current needs.
Inside an environment one or multiple agents are usually instantiated, together with scoreboards, interfaces, coverage collectors and multiple checkers [12].

uvm agent

As we advance through the levels of encapsulation, the Agent represents the next step.

An Agent is usually designed in order to interact with the DUT via an Interface and managing the communication with a Scoreboard and a Checker to control the results obtained.
Two different types of Agent can be identified, namely active and passive Agents.

- **Active Agent**: it is responsible for the generation of the stimulus or sequences that have to be sent to the DUT. Moreover it has to control the timing and conditions under which those stimulus are sent to the DUT. At the same time an Active agent has to observe the response obtained by the DUT after the application of the stimuli and communicate with the Scoreboard;

- **Passive Agent**: differently from the Active it doesn't generate or control stimuli, instead it has the only aim of observing and monitoring the DUT through the interfaces without directly interact with it. The only other action that it has to perform is to communicate with the Scoreboard to control the data observed.

The distinction between an Active and Passive Agent is very clear also in the internal architecture. More in details, the Passive Agent only encapsulates a Monitor, while an Active one encapsulates also a Driver and a Sequencer other than the Monitor. This difference is given by the necessity of stimuli generation through the use of the Driver and the Sequencer, as it will be explained later.
An Agent can be set as active or passive through the control of an internal variable in the developed code. Such variable called `is_active`, and it is set by default as active.

uvm driver

The Driver is an active component class derived from uvm_component and it is used to drive randomized transactions or data to the DUT through the interface. The Driver must retrieve transactions, i.e. data, from the Sequencer and, having knowledge of the DUT functionalities, it must decide the correct timing to apply them to the DUT to adhere to the protocol requirements [5].

uvm monitor

The Monitor is a totally passive component responsible for capturing and monitoring the DUT signals from the interface.
It can have basic checking functionalities to understand if the data observed are correct, even if the majority of them are performed inside the Scoreboard.
In order to observe the data, it has to be connected to the DUT through the interface.

uvm sequencer

The Sequencer is an active component that is responsible for data transactions generation as well as their transmission to the Driver for the execution.
The Sequencer is usually encapsulated inside the agent, but it is common practice to create a sequencer in the Environment, called virtual sequencer or multichannel sequencer. This Sequencer is used to collect multiple sequencers from different agents and activate the correct one at the correct time to send data to the DUT from the right input port. This methodology gives the possibility of controlling multiple sequencer without knowing exactly how they are implemented but only their functionalities, improving code's reusability.
Two different types of sequencers can be identified:

- m_sequencer: is used as a generic handle to give a reference to the sequencer object
- p_sequencer: is a specific handler used to give access to sequencer's properties or methods

uvm sequence

The Sequence is the class that has to define the stimulus pattern for each data item used in the verification process.
If the Sequencer controls timing and ports to check the correct scenarios, the Sequence aims to generate the correct input to stimulate them in the right way.
Input Sequence is usually generated using the random feature of SystemVerilog language. Randomization may be subjected to ad-hoc constraints, so to avoid the generation of non-possible values for specific signals. Thanks to this feature it is possible to test as many different combination of inputs as possible, covering some scenarios that may not come to mind when writing dedicated tests, only.
Once the Sequence has set the data item's boundaries and it has generated the data, it will be taken by the sequencer and it will be transmitted to the Driver to be executed [13].

uvm scoreboard

The Scoreboard is an active class contained in the Environment. It has the main purpose of comparing the output DUT data with the expected output calculated starting from the DUT inputs.
A Scoreboard has three main functions:

- **Reference model or transfer function**: to perform its checker functionalities the Scoreboard needs to reproduce the DUT behaviour and calculate the

expected output data starting from the input ones. The DUT behaviour has to be replicated without knowing its original design to avoid possible common mistakes. The model is usually written in high level languages like C++ or SystemVerilog,

- **Internal Storage**: DUT output needs time to be ready and available, while the reference model is usually an untimed algorithm. Due to that timing discrepancy, the Scoreboard is capable of storing data so that it is able to align the model and DUT output values and perform a meaningful comparison,
- **Comparison logic**: used to control and check the calculated output with the actual DUT one in order to understand if the behaviour of the DUT is correct or not. This checker can be very straightforward with a simple equivalence or very complicated with a complex functional model.

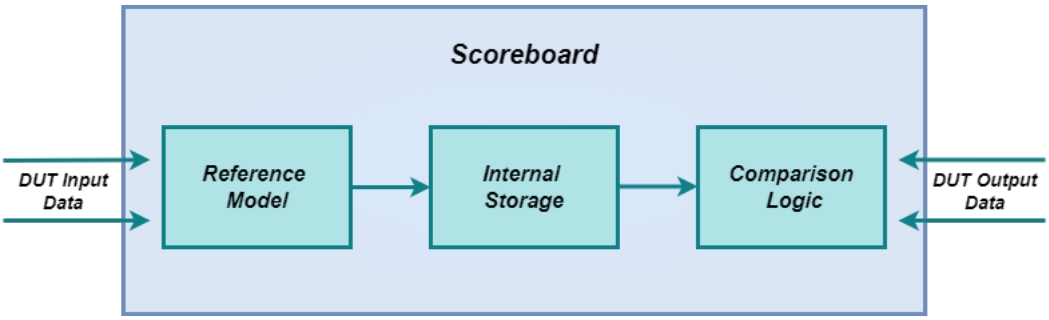

Figure 3.3: UVM Scoreboard

In addition, a Scoreboard can also contain coverage functionalities or be part of a complex hierarchy to determine the correctness of the results.
Scoreboard usually obtains data from a monitor connected to the input of the DUT and from another monitor connected to the DUT output. It is also possible that data doesn't arrive in the expected order, thus leading the Scoreboard to analyze them using an out-of-order logic [13] [14].

uvm interface

The Interface is used as the connection between the DUT and the Agent. The Interface encapsulates all signals needed to let the two entities communicate. The encapsulation simplifies the change of components or DUT in the testbench without compromising the data flow and transactions.
The Interface is used to hide the low level details of signal management, thus simplifying the verification environment and making it easier to be reviewed.

Chapter 4

Testbench Implementation

4.1 Preliminary Choices

In order to accomplish the goals of this thesis, the adopted approach consists of trying to simulate an industrial project using the Metric-Driven Verification (MDV) methodology. Throughout the various steps of the thesis project, I had the valuable opportunity to discuss ideas, doubts and problems with people much more experienced than me. Their guide helped me in finding new strategic choices and alternative approaches to successfully reach the final goal.
Following the four steps cycle of the MDV methodology, the first phase is the Plan definition. During this period, after having studied the I2S protocol, the initial task was to identify, by means of a preliminary analysis, the key features and specifications that the verification process needed to control.
A few key points have been identified:

- **Simple TX**: a transmission where the number of transmitted items doesn't saturate the receiver's FIFO,
- **TX with Overwrite**: a transmission where the amount of data transmitted is larger than the amount of data that can be stored in the FIFO, thus reproducing the scenario where the FIFO is completely filled while additional incoming data is being received,
- **Simple RX**: a reception where the number of received items doesn't empty the FIFO;
- **RX with Underread**: a reception in which the FIFO is read till it becomes empty, after that other additional read operations are attempted,
- **Reset management**: the correct functioning of the reset has to be tested by

ensuring that the logical value '0' is assigned to all signals, with the exception of the SD signal that must be set to Z, i.e., high impedance,

- **Check Channel and Data**: it is important to check that data is correctly generated and redirected to the appropriate channel. Data generated goes in the master unit to be transmitted on channel X and it is effectively transmitted on that channel,
- **Correct Mode**: verify that the protocol performs the right operation, either Transmission or Reception, according to the control signal,
- **Enable signal management**: check that actions are performed only when the enable signal is asserted and that are all interrupted when enable is deasserted.

Once all the key points have been identified and described, the next step consisted of outlining a testbench structure comprised of all the sub-classes needed to carry out the verification process.
The following testbench topology has been chosen:

- Top
  - Test
    - Environment
      - i2s_agent
      - fifo_agent
      - cfg_agent
      - scoreboard

Each agent is active, thus requiring both monitoring and driving capabilities. For this reason, every agent contains a sequencer, a driver and a monitor.
As detailed by the previous list, the choice has been to subdivide the verification environment into three agents having very similar characteristics but each covering a specific aspect of the DUT.
The i2s_agent is responsible for handling the protocol connection to the peripheral. The fifo_agent, on the other hand, is the one controlling the input and output pins of the FIFOs, while the cfg_agent is the one overseeing the system's control signals and ensuring proper configuration settings.
Agents are not directly connected to the DUT, as a dedicated interface is used to manage and organize signals for each agent. This division is performed in order to increase the verification environment's modularity and improving flexibility and maintainability.

In order to better exemplify the proposed testbench structure, a graphic representation is depicted in fig. 4.1.

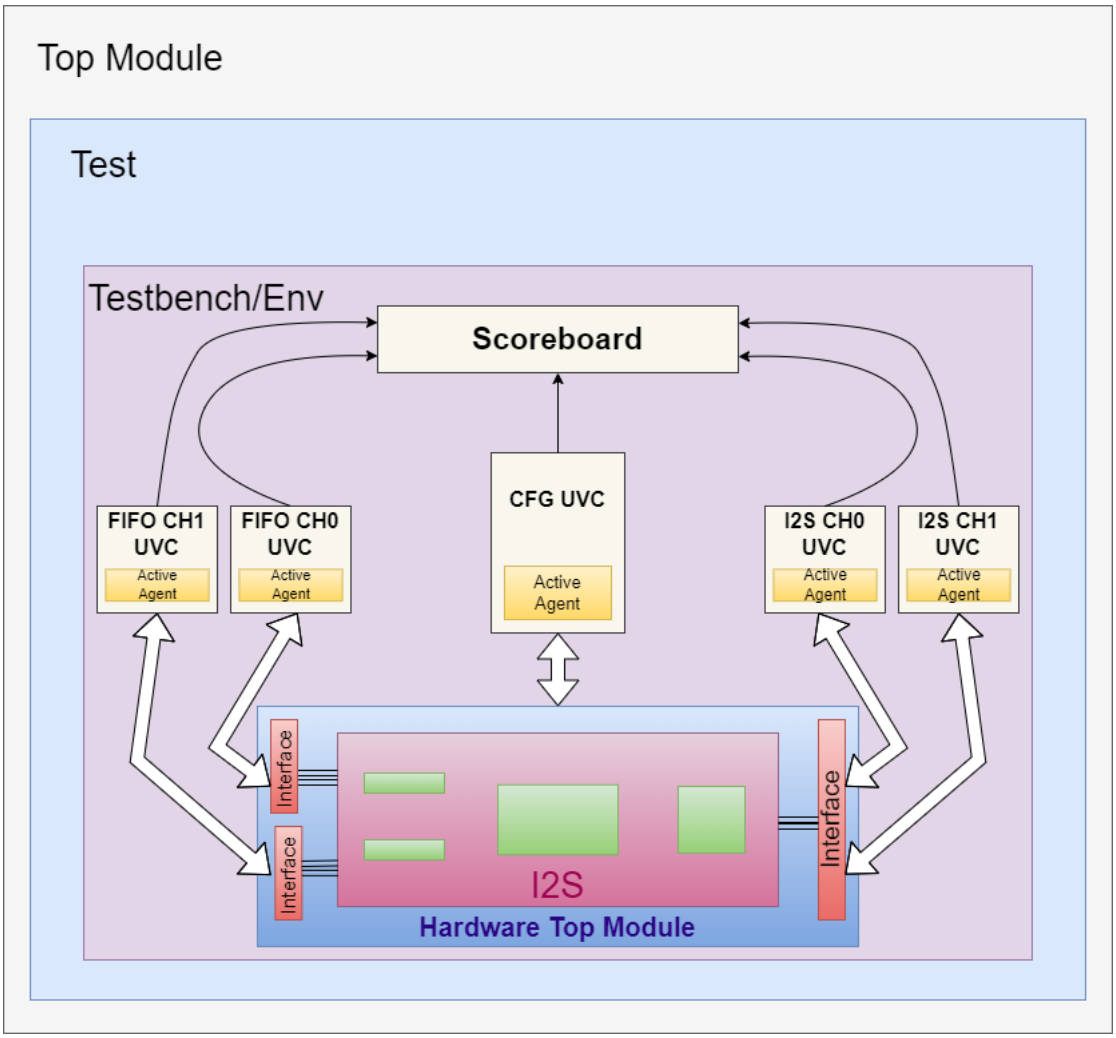

Figure 4.1: UVM Scoreboard

As showed in the figure above, the two FIFOs channels have been managed and controlled separately, each one with its dedicated agent. The I2S has been designed similarly, with channel 0 controlled through agent i2s_ch0 and channel 1 with i2s_ch1.
While the structure is divided into 2 agents for each side, the same thing cannot be said for what concerns the code files.
Thanks to the functionalities offered by SystemVerilog, in order to create the same agent twice with small differences just few lines of code are needed. Through the combination of the get and set methods the two agents can be easily instantiated

simplifying the code, reducing the amount of code lines and increasing the manageability.

The SystemVerilog database is used in order to effectively handle different versions of the same agent. It allows the storage and retrieval of variable values across different testbench components. The access to the database is guaranteed through the use of uvm_config_db class, a class that provides two key functions: set and get. These are, respectively, used to store and retrieve information in and from the database. Both functions have a quite similar implementation, as they need:

- a context: required to know what components can access the database's items,
- an instance name,
- a field name,
- the actual value that has to be stored.

The paths provided in the function fields are often written using wildcards like "*" or "?" in order to enable the selection of multiple possible components using a single path. An example of how the set function can be used is showed in listing 4.1.

Listing 4.1: set method

```
1 fifo_vif_config::set(null, "*.env_tb.*", "vif_fifo_cov", fifo_if_ch0)
```

The introduction of the third agent, cfg_agent, has been another important design choice. This agent, as it will be explained in-depth in the dedicated paragraph, has the main aim of managing the control signals such as reset and enable, and distribute them to the other components.
An alternative approach could have been to integrate the management of these configuration signals inside one of the other two agent components. This, however, would have significantly reduced the modularity of the code, hence why the third agent has been included. Other than that, this decision increases also the readability and comprehensibility of the verification environment.

In order to further clarify the functionalities of the testbench and illustrate the most important features of the code, each file is deeply explained individually. Due to the big similarity between some classes of different agents some sections will be quite similar but, for the sake of completeness, all are reported in the chapter.

4.2 Top Module

The top module is the top level entity that controls and contains everything in the proposed architecture.
This module has the fundamental aim of importing the uvm_pkg and including the uvm_macros in order to guarantee that all the uvm functions and macros are available for the sub components.

Listing 4.2: top module | import and include

```
1  module top;
2
3  // import the UVM library
4      import uvm_pkg::*;
5  // include the UVM macros
6      `include "uvm_macros.svh"
7
8  //import cfg package
9      import cfg_pkg::*;
10 //import fifo package
11     import fifo_pkg::*;
12 //import i2s package
13     import i2s_pkg::*;
14
15 //include the virtual sequencer
16     `include "mc_sequencer.sv"
17 //include virtual sequencer sequences
18     `include "mcseq_lib.sv"
19
20 //include functional coverage
21     `include "coverage.sv"
22
23 //include the scoreboard
24     `include "scoreboard.sv"
25
26 //include the env
27     `include "env.sv"
28
29 //include the test_lib
30     `include "test.sv"
```

Packages for cfg, fifo and i2s blocks are used in order to prevent the presence of large amount of code lines in the top entity, by grouping within them all the include related to the single component resulting in a more comprehensible and immediate code.
Other than to simplify the code, the pkg files is used in order to avoid further modifications in the top module in case of changes in the architecture during advanced steps. Indeed, it is not so unusual that, during the real development of

the testbench, some decisions previously taken could be re-evaluated with possible modification of the components present.
In this case, the only file that has to be modified is the relative pkg, leading to a better organization in working teams.

Another important task performed by the top entity is the construction of the hierarchy. All the sub components are included, namely the environment, the testbench, the scoreboard, the coverage calculator, the virtual sequencer, the virtual sequencer sequences and, finally, the DUT, named in the testbench as i2s_master_txrx.
The connection of the DUT to the verification environment is performed through the use of interfaces, fifo_if, cfg_if and i2s_if. All the signals connections are explicitly declared in the code, the choice has been to use an explicit declaration of the connections in order to have a better understanding of the environment even with a slightly longer code.

Listing 4.3: top module | DUT and Interfaces

```
1  //FIFO Interface to DUT
2  fifo_if      fifo_if_ch0(clock, reset, enable_top, tx_nrx_top);
3  fifo_if      fifo_if_ch1(clock, reset, enable_top, tx_nrx_top);
4
5  //i2s Interface
6  i2s_if       i2s_if(sck, reset, enable_top, tx_nrx_top);
7
8  cfg_if       cfg_if(clock, reset, enable_top, tx_nrx_top);
9
10 i2s_master_txrx dut(.clk(clock),
11                     .rst_b(reset),
12
13                     .cfg_tx_nrx(tx_nrx_top),
14                     .cfg_i2s_en(enable_top),
15
16                     //FIFO ch0 interface
17                     .ch0_data_tx(fifo_if_ch0.data_tx),
18                     .ch0_data_tx_put_en(fifo_if_ch0.data_tx_put_en),
19                     .ch0_fifo_underrun(fifo_if_ch0.fifo_underrun),
20                     .ch0_fifo_overrun(fifo_if_ch0.fifo_overrun),
21                     .ch0_data_rx(fifo_if_ch0.data_rx),
22                     .ch0_data_rx_get_en(fifo_if_ch0.data_rx_get_en),
23
24                     //FIFO ch1 interface
25                     .ch1_data_tx(fifo_if_ch1.data_tx),
26                     .ch1_data_tx_put_en(fifo_if_ch1.data_tx_put_en),
27                     .ch1_fifo_underrun(fifo_if_ch1.fifo_underrun),
28                     .ch1_fifo_overrun(fifo_if_ch1.fifo_overrun),
29                     .ch1_data_rx(fifo_if_ch1.data_rx),
```

```
30                          .ch1_data_rx_get_en(fifo_if_ch1.data_rx_get_en),
31
32                          //I2S interface output
33                          .SCK(sck),
34                          .WS(i2s_if.ws),
35                          .SD(i2s_if.sd)
36                         );
```

Then, as aforementioned, the uvm database is exploited through the use of uvm_config_db class and its set and get methods in order to store required values that must be passed on to other components.

Listing 4.4: top module | set functions

```
1 i2s_vif_config::set(null, "*.env_tb.*", "vif_i2s_cov", i2s_if);
2 fifo_vif_config::set(null, "*.env_tb.agent_fifo_ch0.*","vif_fifo",
    fifo_if_ch0);
```

As it is visible in the code snippet above, the set method is used to assign to the correct agent the correspondent channel interface and, consequently, the contained signals.

The use of wildcards "*" helps to reference multiple components at the same time: the "*" replaces a part of path that can be filled by any other path that correctly matches the fixed part written before or after the wildcard. This feature helps in dealing with complex structures and deep architectures avoiding the repetition of multiple lines, reducing the code length and increasing the readability.

Finally, the last operation performed by the top module is to run the test. This is done through the use of the method run_test() that starts the simulation of the testbench and constructs a uvm_root object. Moreover, it gets the test name that has to be run and it constructs the test object, too. Lastly it starts the phases system, allowing the testbench to run.

4.3 Test

The test class is the one in charge of building the architecture needed to verify a specific feature of the DUT. Due to the necessity of trying out multiple features, the test file consists of a library of tests. In this file, different tests were written in order to adapt the characteristics of the testbench to the features that must be tested.
The first test written is the base_test class, from which all the other tests will be derived. This test contains the instantiation of the environment and config_fifo (that contains configurations for the fifo_agent), and the set of all the phases useful with meaningful checks.

Listing 4.5: test class | check_phase and end_of_elaboration_phase

```
//CHECK_PHASE
    function void base_test::check_phase(uvm_phase phase);
    check_config_usage();
        apply_config_settings();
    endfunction : check_phase
//END_OF_ELABORATION_PHASE
    function void base_test::end_of_elaboration_phase(uvm_phase phase
    );
        uvm_top.print_topology();
    endfunction : end_of_elaboration_phase
```

Among the generic tasks, there is the creation of the environment class in the build_phase, some configurations checks like the use of check_config_usage() and apply_config_settings() methods in the check_phase, an useful debug control like print_topology in the end_of_elaboration_phase and the set of the drain time at 200 ns with the set_drain_time method in the run_phase.

Listing 4.6: test class | run_phase

```
//RUN_PHASE
    task base_test::run_phase(uvm_phase phase);
        super.run_phase(phase);
        phase.phase_done.set_drain_time(this, 200ns);
    endtask : run_phase
```

To explain the use of the drain time, it is necessary to first introduce another important topic: *objections*.
The objection mechanism is the method used to coordinate the communications and the work done by different modules. When a component starts working during the run_phase, an objection is raised in order to point out that an operation has started; when the operation is concluded the objection is dropped. Once all the objections are dropped the system can change phase, exiting from the run_phase. Drain time is a time added at the end of all the operations once all the objections have been dropped, so that it is possible to ensure that all the transactions have been completed and nothing is still running. To summarize, drain time is a fixed amount of time that is used to check that no more objections are raised when all the current raised ones have been dropped.

From the base_test seven tests are derived:

- tx_test
- tx_last_test
- tx_over_test

- rx_test
- rx_last_test
- rx_over_test
- not_rst_test

Each test has been specifically written for a test scenario that had to be verified, configuring in the virtual sequencer the correspondent ad hoc written sequence to inject input values that are instrumental to create the test scenario.
The class tx_test has been created in order to verify a simple transmission, that is, a data packet transmitted from the FIFO side to the I2S side without creating an overload condition in the FIFO.
The FIFO overload scenario for the transmission is tested in the tx_over_test class, where a higher number of data is expected to be transmitted in a short time frame to control the correct management of the overload.
A similar motivation resides behind the creation of the rx_test and rx_over_test, two tests that have been created to verify the correctness of the reception mechanism, i.e., a transmission of data from the i2s to the FIFO side, either without any underrun condition in rx_test and with an underrun condition in rx_over_test.
The not_rst_test has been created in order to control the behaviour of the DUT in case of the absence of a toggle on the reset signal. Lastly, the remaining two sequences tx_last_test and rx_last_test has been written for coverage purposes and will be explained in details in the coverage chapter.
In each of the described tests, the uvm_config_wrapper::set method is used to select the virtual sequencer and the sequence to be transmitted on it. The rx_over_test with the rx_over_v_seq applied in the virtual sequencer identified by the path "env_tb.mc_seqr.run_phase" is reported in listing 4.7.

Listing 4.7: test class | rx_over_test

```
1  class rx_over_test extends base_test;
2
3      `uvm_component_utils(rx_over_test)
4
5      function new(string name, uvm_component parent);
6          super.new(name, parent);
7      endfunction
8
9      extern function void build_phase(uvm_phase phase);
10 endclass : rx_over_test
11
12     //BUILD_PHASE
13     function void rx_over_test::build_phase(uvm_phase phase);
14
```

```
15          //virtual sequencer
16          uvm_config_wrapper::set(this, "env_tb.mc_seqr.run_phase", "
    default_sequence", rx_over_v_seq::type_id::get());
17
18          super.build_phase(phase);
19      endfunction : build_phase
```

4.4 Environment

The testbench environment file is called env.sv. On this level, agents of both channels are instantiated along with configuration classes, the virtual sequencer, the scoreboard and coverage class.
Moreover, it is important to underline the use of set and create methods in the build_phase to construct the testbench architecture through the single components and set the variables in the uvm database. In listing 4.8, a few lines of the build_phase are displayed. In line 2, the uvm_config_db with set method is used to assign the value 0 to the variable ch_id of the class agent_i2s_ch0; similarly, in line 3 value 1 is assigned to ch_id of class agent_i2s_ch1. In the other lines, the creation of the class objects using the method create can be observed.

Listing 4.8: environment class | build_phase

```
1   //set of ch_id into the correspondent agent
2   uvm_config_db#(int)::set(this, "*.agent_i2s_ch0", "ch_id", 0);
3   uvm_config_db#(int)::set(this, "*.agent_i2s_ch1", "ch_id", 1);
4
5   //create of the i2s_agent
6   agent_i2s_ch0 = i2s_agent::type_id::create("agent_i2s_ch0", this);
7   agent_i2s_ch1 = i2s_agent::type_id::create("agent_i2s_ch1", this);
8
9   //create of the mc_sequencer
10  mc_seqr = mc_sequencer::type_id::create("mc_seqr", this);
11
12  //create of the scoreboard
13  scoreb = scoreboard::type_id::create("scoreb", this);
```

Afterwards, the connect_phase is used in order to generate the connection between the virtual sequencer and the agents of the various components. Then, the scoreboard connection with the FIFO agents and I2S agents is created through the use of analysis ports.

Analysis port is a class used to create connections between components instantiated at the same architectural level, in the situation displayed the environment level. The components link is performed through the use of the connect method as it can be seen in listing 4.9

Listing 4.9: environment class | scoreboard connections

```
1     agent_fifo_ch0.mon_fifo.fifo_write_port.connect(scoreb.
      sb_fifo_ch0);
2     agent_i2s_ch0.mon_i2s.i2s_write_port.connect(scoreb.sb_i2s_ch0);
```

4.5 Virtual Sequencer

The virtual sequencer is a special component used to coordinate the execution of multiple sequencers of different components. The file is called mc_sequencer.sv.
In the developed code, the virtual sequencer is called mc_sequencer (the name derives from the contraction of multichannel sequencer which is an alternative name of the virtual sequencer).
It contains the instantiation of all the agents' sequencer, which will be later recalled and used in the virtual sequence class mcseq_lib to express on which sequencer each sequence has to be run.

Listing 4.10: mc_sequencer | code snippet

```
1  class mc_sequencer extends uvm_sequencer;
2      `uvm_component_utils(mc_sequencer)
3
4      fifo_sequencer   fifo_seqr0, fifo_seqr1;
5      i2s_sequencer    i2s_seqr0, i2s_seqr1;
6      cfg_sequencer    cfg_seqr;
7
8      function new(string name, uvm_component parent);
9          super.new(name, parent);
10     endfunction : new
11 endclass : mc_sequencer
```

4.6 Virtual Sequences

Virtual sequences are contained in a file called mcseq_lib.sv. It is the collection of the sequences used by the sequencers contained in the mc_sequencer.
At first a base sequence called and base_mcseq is created. This sequence is structured in order to derive all the other required sequences. The base sequence contains the object handler of all the single sequences of the sequencers collected in the virtual sequencer.
Then, in the pre_body and post_body phases the objections are managed. As described before, the objections methodology is used to understand if there are some sequences that are active or not in each time instant. The pre_body phase is activated before the execution of the body of each derived sequence raising the

objection, and after the conclusion of the body task the post_body is performed, dropping the objection.

Listing 4.11: base_mcseq | pre_body and post_body

```
1
2  //PRE_BODY
3      task  base_mcseq::pre_body();
4          uvm_phase phasef;
5
6          phasef = get_starting_phase();
7
8          if (phasef != null) begin
9              phasef.raise_objection(this, get_type_name());
10             `uvm_info(get_type_name(), "raise objection virtual
       sequence", UVM_MEDIUM)
11         end
12     endtask : pre_body
13
14     //POST_BODY
15     task  base_mcseq::post_body();
16         uvm_phase phasef;
17
18         phasef = get_starting_phase();
19
20         if(phasef != null) begin
21             phasef.drop_objection(this, get_type_name());
22             `uvm_info(get_type_name(), "drop objection virtual
       sequence", UVM_MEDIUM)
23         end
24     endtask : post_body
```

Once the base_mcseq has been defined, all the other sequences are derived from that.
A total of seven sequences are derived:

- tx_v_seq
- tx_last_v_seq
- tx_over_v_seq
- rx_v_seq
- rx_last_v_seq
- rx_over_v_seq
- not_rst_v_seq

As it can be deducted from the names chosen, there is an obvious parallelism between the tx and rx sequences having as main difference the data direction.
The tx_v_seq and rx_v_seq are used to perform a transmission or reception with a simple read and write of the FIFOs, without having complex scenarios like overwrite or underread in the FIFOs.
The tx_over_v_seq and rx_over_v_seq are the two virtual sequences used to evaluate the scenarios with overwrite and underrun. The main difference, other than the change of sequences used, is the waiting time set at 16000 ns. This time has been estimated empirically knowing a priori the number of cycles that each sequence needs to complete its operations. Sequences tx_last_v_seq and rx_last_v_seq have been generated after gathering the first coverage data. These virtual sequences are used to perform the fifo_last_seq and i2s_last_seq. As explained in the chapter 5, those are constrained sequences that have been generated to cover the few inputs not randomly generated. With the execution of those sequences the expected coverage is 100%.

The tx_v_seq is shown in listing 4.12 After the use of cfg_raise_enable_rst sequence to raise the enable signal and power up the unit, a sequence is started on each sequencer in the same instant of time using a fork structure. The fork is concluded using a join_none and then waiting 5000 ns. This time is needed in order to let the sequences perform the operations on the master unit.
After the waiting time the cfg_drop_enable_rst sequence is started on the cfg_sequencer in order to drop the enable signal and change the state to IDLE.

Listing 4.12: tx_v_seq | body task

```
1  //BODY
2       virtual task body();
3          `uvm_info("base_mcseq", "execute tx_v_seq of
       virtual_sequencer", UVM_LOW)
4
5          `uvm_do_on(cfg_raise_enable_rst, p_sequencer.cfg_seqr)
6
7          fork
8              `uvm_do_on(fifo_wr_seq, p_sequencer.fifo_seqr0)
9              `uvm_do_on(i2s_rd_seq, p_sequencer.i2s_seqr0)
10             `uvm_do_on(fifo_wr_seq, p_sequencer.fifo_seqr1)
11             `uvm_do_on(i2s_rd_seq, p_sequencer.i2s_seqr1)
12             `uvm_do_on(cfg_write, p_sequencer.cfg_seqr)
13         join_none;
14
15         #5000;
16
17         `uvm_do_on(cfg_drop_enable_rst, p_sequencer.cfg_seqr)
18     endtask : body
```

4.7 Configuration files

The following files:

- config_i2s.sv
- config_fifo.sv

are configuration files. The main aim is setting the agent and environment configurations. As usual, the creation of these classes has been done for the sake of modularity and encapsulation: in this way it is more clear where one information can be found and retrieved. Due to the simplicity of the architecture, the active_passive variable of the agents is contained and set only.

Listing 4.13: config_fifo active_passive

```
1    //fifo_agent is_active
2    uvm_active_passive_enum fifo_active = UVM_ACTIVE;
```

4.8 Define_pkg

The define_pkg contains all the constants used in the testbench. The keyword used to declare a constant gives its name to the file.
Four constants are contained:

- WORD_LEN_LOG2: which is the base 2 logarithm of the word length;
- FIFO_ADDR_LEN: is the FIFO address' number of bits;
- WORD_LEN: represents the word length;
- FIFO_MEM_SIZE: is the maximum number of item that can be stored in a FIFO;

4.9 Scoreboard

The UVM Scoreboard is a crucial component used to check the correct functioning of the DUT. The scoreboard class used in the testbench is shown in listing 4.14. The idea was to create two queues for each channel to store and compare data at DUT's input and output, ensuring that data remains unaltered during the transmission, as required for a correct functionality of the I2S protocol.
Due to the timing delays applied by the transmission, the queue at the input has to maintain data till the transmission is completed and data is ready at the output.

From line 17 to 20, the queues are created. The queue fifo_q stores data at the FIFO side, while i2s_q manages data at the I2S side. Queues have been defined as dynamic to manage the unpredictable number of data items that must be stored by the input queue.
The scoreboard code can be divided in five parts:

- write_fifo_ch0
- write_fifo_ch1
- write_i2s_ch0
- write_i2s_ch
- report_phase

The four "write" functions differ on the channel managed and the side of the DUT that control.
To illustrate the code in details, the write_fifo_ch0 function is explained.
This function takes as argument a fifo_packet, which represents the data item present at the FIFO side. According to the working mode, checked at line 60 of listing 4.14, the packet is simply added to the corresponding queue using the push_back method (line 61 of listing 4.14) or compared to the other DUT side packet.
In the function taken as example, if the DUT is in RX mode, the first step consists of checking the i2s side queue size. If it is not empty, the first element is taken and its data field is compared with the first one saved in the FIFO queue. According to the comparison result the correspondent counter is incremented, counting the number of successful and failed receptions. In order to prepare the data for the next check, the data compared is popped out (line 79 of listing 4.14) at the end of the process.
Finally, in the report phase the transmission results are listed to offer a meaningful summary on the number of packets elaborated.

Listing 4.14: scoreboard

```
1  class scoreboard extends uvm_scoreboard;
2
3      `uvm_analysis_imp_decl(_fifo_ch0)
4      `uvm_analysis_imp_decl(_fifo_ch1)
5
6      `uvm_analysis_imp_decl(_i2s_ch0)
7      `uvm_analysis_imp_decl(_i2s_ch1)
8
9      uvm_analysis_imp_fifo_ch0#(fifo_packet, scoreboard) sb_fifo_ch0;
10     uvm_analysis_imp_fifo_ch1#(fifo_packet, scoreboard) sb_fifo_ch1;
```

```
11
12      uvm_analysis_imp_i2s_ch0#(i2s_packet, scoreboard) sb_i2s_ch0;
13      uvm_analysis_imp_i2s_ch1#(i2s_packet, scoreboard) sb_i2s_ch1;
14
15      `uvm_component_utils(scoreboard)
16
17      fifo_packet fifo_q_ch0[$];
18      fifo_packet fifo_q_ch1[$];
19      i2s_packet i2s_q_ch0[$];
20      i2s_packet i2s_q_ch1[$];
21
22
23      //scoreboard statistics
24      int success_tx, fail_tx = 0;
25      int success_rx, fail_rx = 0;
26
27      int num_packet_i2s_0, num_packet_i2s_1 = 0;
28      int num_packet_fifo_0, num_packet_fifo_1 = 0;
29
30      int check = 0;
31
32      function new (string name, uvm_component parent);
33          super.new(name, parent);
34
35          //new TLM ports
36          sb_fifo_ch0 = new("sb_fifo_ch0", this);
37          sb_fifo_ch1 = new("sb_fifo_ch1", this);
38
39          sb_i2s_ch0 = new("sb_i2s_ch0", this);
40          sb_i2s_ch1 = new("sb_i2s_ch1", this);
41      endfunction : new
42
43      extern virtual function void write_fifo_ch0(fifo_packet packet);
44      extern virtual function void write_i2s_ch0(i2s_packet packet);
45      extern virtual function void write_fifo_ch1(fifo_packet packet);
46      extern virtual function void write_i2s_ch1(i2s_packet packet);
47      extern function void report_phase(uvm_phase phase);
48
49  endclass :scoreboard
50
51
52      //WRITE_FIFO_CH0
53      function void scoreboard::write_fifo_ch0(fifo_packet packet);
54
55          i2s_packet  i2s_p;
56          fifo_packet fifo_p;
57          $cast(fifo_p, packet.clone());
58          num_packet_fifo_0++;
59
```

```
60          if(fifo_p.tx_nrx) begin
61              fifo_q_ch0.push_back(fifo_p);
62              `uvm_info(get_type_name(), $sformatf("ADDED 0 fifo packet
     to Scoreboard Queue 0: \n%s \n\n", fifo_p.sprint()), UVM_HIGH);
63          end
64          else begin
65              if (i2s_q_ch0.size() == 0 ) begin
66                  `uvm_info(get_type_name(), $sformatf("Scoreboard
    Error EMPTY QUEUE on channel 0"), UVM_HIGH);
67                  return;
68              end
69
70              i2s_p = i2s_q_ch0[0];
71              if (fifo_p.data == i2s_p.sd) begin
72                 `uvm_info(get_type_name(), $sformatf("Scoreboard
    CORRECT 0 Check: Channel_0 Packet\n%s \n\n", i2s_p.sprint()),
    UVM_MEDIUM)
73                  success_rx++;
74              end
75              else begin
76                  `uvm_info(get_type_name(), $sformatf("Scoreboard
    ERROR NOT MATCH: Channel_0 Packet \n%s \n\n", i2s_p.sprint()),
    UVM_MEDIUM)
77                  fail_rx++;
78              end
79              i2s_q_ch0.pop_front();
80          end
81
82      endfunction : write_fifo_ch0
83
84
85      //WRITE_FIFO_CH1
86      function void scoreboard::write_fifo_ch1(fifo_packet packet);
87
88          i2s_packet  i2s_p;
89          fifo_packet fifo_p;
90          $cast(fifo_p, packet.clone());
91          num_packet_fifo_1++;
92
93          if(fifo_p.tx_nrx) begin
94              fifo_q_ch1.push_back(fifo_p);
95              `uvm_info(get_type_name(), $sformatf("ADDED 1 fifo packet
     to Scoreboard Queue 1: \n%s \n\n", fifo_p.sprint()), UVM_HIGH);
96          end
97          else begin
98              if (i2s_q_ch1.size() == 0 ) begin
99                  `uvm_info(get_type_name(), $sformatf("Scoreboard
    Error EMPTY QUEUE on channel 1"), UVM_HIGH);
100                 return;
```

```
101             end
102
103             i2s_p = i2s_q_ch1[0];
104             if (fifo_p.data == i2s_p.sd) begin
105                 `uvm_info(get_type_name(), $sformatf("Scoreboard
    CORRECT 1 Check: Channel_1 Packet\n%s \n\n", i2s_p.sprint()),
    UVM_MEDIUM)
106                 success_rx++;
107             end
108             else begin
109                 `uvm_info(get_type_name(), $sformatf("Scoreboard
    ERROR NOT MATCH: Channel_1 Packet \n%s \n\n", i2s_p.sprint()),
    UVM_MEDIUM)
110                 fail_rx++;
111             end
112             i2s_q_ch1.pop_front();
113         end
114
115     endfunction : write_fifo_ch1
116
117
118     //WRITE_I2S_CH0
119     function void scoreboard::write_i2s_ch0(i2s_packet packet);
120
121         i2s_packet i2s_p;
122         fifo_packet fifo_p;
123         $cast(i2s_p, packet.clone());
124         num_packet_i2s_0++;
125
126         if(i2s_p.tx_nrx) begin
127             if (fifo_q_ch0.size() == 0 ) begin
128                 `uvm_info(get_type_name(), $sformatf("Scoreboard
    Error EMPTY QUEUE on channel 0"), UVM_HIGH);
129                 return;
130             end
131
132             fifo_p = fifo_q_ch0[0];
133             if (fifo_p.data == i2s_p.sd) begin
134                 `uvm_info(get_type_name(), $sformatf("Scoreboard
    CORRECT 0 Check: Channel_0 Packet\n%s \n\n", fifo_p.sprint()),
    UVM_MEDIUM)
135                 success_tx++;
136             end
137             else begin
138                 `uvm_info(get_type_name(), $sformatf("Scoreboard
    ERROR NOT MATCH: Channel_0 Packet \n%s \n\n", fifo_p.sprint()),
    UVM_MEDIUM)
139                 fail_tx++;
140             end
```

```
141             fifo_q_ch0.pop_front();
142         end
143         else begin
144             i2s_q_ch0.push_back(i2s_p);
145             `uvm_info(get_type_name(), "Added i2s packet to
    Scoreboard Queue 0", UVM_HIGH);
146         end
147
148     endfunction : write_i2s_ch0
149
150
151     //WRITE_I2S_CH1
152     function void scoreboard::write_i2s_ch1(i2s_packet packet);
153
154         i2s_packet i2s_p;
155         fifo_packet fifo_p;
156         $cast(i2s_p, packet.clone());
157         num_packet_i2s_1++;
158
159         if(i2s_p.tx_nrx) begin
160             if (fifo_q_ch1.size() == 0 ) begin
161                 `uvm_info(get_type_name(), $sformatf("Scoreboard
    Error EMPTY QUEUE on channel 1"), UVM_HIGH);
162                 return;
163             end
164
165             fifo_p = fifo_q_ch1[0];
166             if (fifo_p.data == i2s_p.sd) begin
167                `uvm_info(get_type_name(), $sformatf("Scoreboard
    CORRECT 1 Check: Channel_1 Packet\n%s \n\n", fifo_p.sprint()),
    UVM_MEDIUM)
168                 success_tx++;
169             end
170             else begin
171                 `uvm_info(get_type_name(), $sformatf("Scoreboard
    ERROR NOT MATCH: Channel_1 Packet \n%s \n\n", fifo_p.sprint()),
    UVM_MEDIUM)
172                 fail_tx++;
173             end
174              fifo_q_ch1.pop_front();
175         end
176         else begin
177             i2s_q_ch1.push_back(i2s_p);
178             `uvm_info(get_type_name(), "Added i2s packet to
    Scoreboard Queue 1", UVM_HIGH);
179         end
180
181     endfunction : write_i2s_ch1
182
```

```
    //REPORT_PHASE
    function void scoreboard::report_phase(uvm_phase phase);
        num_packet_i2s_0 = num_packet_i2s_0 + num_packet_i2s_1;
        num_packet_fifo_0 = num_packet_fifo_0 + num_packet_fifo_1;
        `uvm_info(get_type_name(), $sformatf("\n
        @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        \nReport:\n\n  SCOREBOARD: \n SEQ_ITEM analized: \n
        FIFO_PACKET: %d         I2S_PACKET: %d \n\n
        SUCCESS TX : %d         FAIL TX:     %d \n
        SUCCESS RX : %d         FAIL RX:    %d\n\n
        @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@",
        num_packet_fifo_0 , num_packet_i2s_0, success_tx, fail_tx, success_rx, fail_rx), UVM_LOW)
    endfunction : report_phase
```

4.10 FIFO

In this section all files related to the FIFO are analyzed in depth.

4.10.1 FIFO package

This file is called fifo_pkg.sv, and it is used to group all the include related to FIFO. After the uvm_macros, all the single components are included. It is important to underline the order of the include which is not random, as the components are listed using a bottom-up approach. When the compilation starts, all the single files are analyzed and synthesized. This approach is used to ensure no compilation problems given by file dependencies arise.
Moreover, a typedef is present in the file. This keyword is used to avoid the repetition of long code parts in order to use a uvm database's call related to the fifo_interface, as it can be seen in **??**.

Listing 4.15: fifo_pkg | include and typedef

```
    typedef uvm_config_db#(virtual fifo_if) fifo_vif_config;

    `include "fifo_sequencer.sv"
    `include "fifo_seq.sv"
    `include "fifo_driver.sv"
    `include "fifo_monitor.sv"

    `include "fifo_agent.sv"
```

4.10.2 FIFO agent

File fifo_agent.sv contains the fifo_agent. It contains the declaration and instantiation of the sub-components: monitor, driver and sequencer. The build_phase is dedicated to the creation of the hierarchy.
The components inside an agent change according to it being active or passive. As a consequence, the first check done inside the build_phase is the control of the uvm_active_passive_enum fifo_active value contained in the config_fifo.sv file. In case of a passive value an uvm_error is triggered with a debug message displayed as output. It is very important that the uvm_error shows an explanatory statement on the reason of the error in order to make the debug work easier and faster.

Listing 4.16: fifo_agent | active_passive check and components create

```
if (!`obj_config(config_fifo)::get(this, get_full_name(), "
    config_fifo", fifo_cfg))
            `uvm_error("NO_CFG", {"configuration object is not set
    properly for: ", get_full_name()})
        else
            `uvm_info(get_full_name(), "ok get::cfg", UVM_HIGH)

        mon_fifo = fifo_monitor::type_id::create("mon_fifo", this);
        if(fifo_cfg.fifo_active == UVM_ACTIVE) begin
            seqr_fifo = fifo_sequencer::type_id::create("seqr_fifo",
    this);
            drv_fifo = fifo_driver::type_id::create("drv_fifo", this)
    ;
        end
```

In the connect_phase, after the check of the active_passive value, the connection of the driver with the sequencer through the seq_item_port is implemented.

Listing 4.17: fifo_agent | driver - sequencer connection

```
 if(fifo_cfg.fifo_active == UVM_ACTIVE) begin
        //Connect driver and seqr
        drv_fifo.seq_item_port.connect(seqr_fifo.seq_item_export);
 end
```

4.10.3 FIFO driver

The fifo_driver has the fundamental goal of sending data to the DUT at the correct time in order to simulate a real situation where the DUT has to work.
Three tasks must be highlighted in the code:

- run_phase

- send_packet
- receive_packet

The run_phase is the main between those three, as the other two represent the two alternative situations in which the driver has to work: a transmission or a reception.
In the run_phase, a few timing controls are used to synchronize the driver with the rest of the environment and then, according to the value of tx_nrx (that represents the mode in which the driver is working: 0 for RX, 1 for TX), one of the two remaining tasks is selected.
The assignment at line 3 is performed in order to avoid strange values on data_tx in case of a transmission.

Listing 4.18: fifo_driver | run_phase

```
1  task fifo_driver::run_phase(uvm_phase phase);
2
3          vif_fifo.data_tx = 0;
4
5          forever begin
6              wait(vif_fifo.enable);
7              if(vif_fifo.rst_l)
8                  @(posedge vif_fifo.clk)
9                  @(posedge vif_fifo.clk)
10                 if(vif_fifo.tx_nrx)
11                     send_packet();          //task tx
12                 else
13                     receive_packet();       //task rx
14             else
15                 `uvm_info(get_full_name(), "Reset low", UVM_MEDIUM)
16         end
17
18     endtask : run_phase
```

The send_packet task has been designed in order to manage a transmission. Given the fact that, in a transmission, the fifo agent is the one that has to transmit data, a data item is taken from the sequencer using the get_next_item method and transmits data to the FIFO interface. Finally, the driver let the sequencer knows that the transaction is concluded by means of the item_done method. At the next positive edge of the clock, both data_tx and data_tx_put_en are reset to 0.

Listing 4.19: fifo_driver | send_packet - data transmission

```
1      //get new data_item from seqr
2      seq_item_port.get_next_item(req);
3
```

```
4       //drive data_item and record the transaction
5       vif_fifo.data_tx                = req.data;
6       vif_fifo.data_tx_put_en         = 1'b1;
7
8       // Communicate item done to the sequencer
9       seq_item_port.item_done();
10
11      @(posedge vif_fifo.clk)
12      vif_fifo.data_tx                = 'h0;
13      vif_fifo.data_tx_put_en         = 1'b0;
```

When in receive_packet, instead, the task driver is set to correctly manage a reception. In order to achieve that, after the get_next_item method to get an item from the sequencer, the data field is set to the value read by the interface. After that the transaction is concluded with the item_done method called and the data_rx_get_en is set for one clock cycle to report that a data has been received.

Listing 4.20: fifo_driver | receive_packet - data reception

```
1       //get new data_item from seqr
2       seq_item_port.get_next_item(req);
3
4       req.data = vif_fifo.data_rx;
5
6       // Communicate item done to the sequencer
7       seq_item_port.item_done();
8
9       vif_fifo.data_rx_get_en     = 1'b1;
10      @(posedge vif_fifo.clk)
11      vif_fifo.data_rx_get_en         = 1'b0;
```

4.10.4 FIFO interface

The FIFO interface is called fifo_if. It contains all the signals needed to connect the fifo_agent with the DUT.
Signals are divided according to their direction: data_tx, data_tx_put_en, data_rx_get_en are used to drive data from the testbench to the DUT; instead data_rx is used to drive data from DUT to the testbench.
Then, there are two signals asserted, respectively, in case of an underrun or a overrun of the FIFOs: fifo_underrun and fifo_overrun.

Listing 4.21: fifo interface | signals

```
1   //from if to DUT
2       logic [`WORD_LEN-1 : 0]     data_tx;
3       bit                         data_tx_put_en;
4       bit                         data_rx_get_en;
```

```
5
6       //from DUT to if
7       logic [`WORD_LEN -1 : 0]     data_rx;
8
9       bit                          fifo_underrun;
10      bit                          fifo_overrun;
11
12      property enable_txnrx;
13         @(posedge clk) enable |-> rst_l;
14      endproperty
15
16      assert property (enable_txnrx);
```

Last lines are related to coverage and assertions and will be explained in chapter 5.

4.10.5 FIFO monitor

The fifo_monitor is used to look at the values that pass through the fifo_agent. It has an analysis port used for TLM connection to the Scoreboard.
The connection_phase is used to control the correct configuration of the virtual interface and the config_fifo. In case of connection errors or not correct instantiations, an uvm_error is triggered giving a self explanatory message as output.

Listing 4.22: fifo monitor | connect_phase

```
1  function void fifo_monitor::connect_phase(uvm_phase phase);
2      // get virtual interface
3      if (!fifo_vif_config::get(this, get_full_name(),"vif_fifo",
       vif_fifo))
4
5          `uvm_error("NOVIF",{"virtual interface must be set for: ",
       get_full_name(),".vif_fifo"})
6
7      //get config_obj
8      if (!`obj_config(config_fifo)::get(this, get_full_name(), "
       config_fifo", fifo_cfg))
9
10         `uvm_error("NO_CFG_OBJ", {"configuration object is not set
       properly for: ", get_full_name()})
11     else
12         `uvm_info(get_full_name(), "ok get::cfg", UVM_HIGH)
13     endfunction: connect_phase
```

Due to the passive characteristic of the monitor class, the run_phase has two main aims:

- reading data,
- sending data to the scoreboard.

As it can be seen in listing 4.23, the run_phase is controlled by a forever loop followed by a wait condition activated by the reset signal of the FIFO interface (line 3). This condition is set in order to let the read of data start only when the reset signal is deactivated. The check at line 5 is performed in order to capture the data correctly. Indeed, data is synchronized with one of the two signals: data_tx_put_en or data_rx_get_en, respectively, in case of TX or RX. The following lines are used to create the data_item with the characteristics of the fifo_packet and assigning the data read to the correct field. Line 18 is used to display the data read, useful mostly for debug purposes.
Lastly, line 21 is the write method of the analysis port associated to the monitor - scoreboard connection.

Listing 4.23: fifo monitor | run_phase

```
1  task fifo_monitor::run_phase(uvm_phase phase);
2      forever begin
3      wait(vif_fifo.rst_l);
4          @(posedge vif_fifo.clk)
5          if(vif_fifo.data_tx_put_en || vif_fifo.data_rx_get_en) begin
6
7              // Create Packet
8              data_write = fifo_packet::type_id::create("data_write",
       this);
9
10             data_write.tx_nrx = vif_fifo.tx_nrx;
11
12             if (vif_fifo.tx_nrx && vif_fifo.data_tx_put_en)
13                 data_write.data = vif_fifo.data_tx;
14             else if (!vif_fifo.tx_nrx && vif_fifo.data_rx_get_en)
15                 data_write.data = vif_fifo.data_rx;
16
17             @(posedge vif_fifo.clk)
18             `uvm_info(get_type_name(), $sformatf(" Packet collected
       :\n%s", data_write.sprint()), UVM_LOW)
19
20             //send packet to scoreboard via TLM write()
21             fifo_write_port.write(data_write);
22         end
23     end
24 endtask : run_phase
```

4.10.6 FIFO packet

FIFO packet represents the sequence item used by the fifo_agent. It defines the data fields that will be present in each transaction.
In a fifo_packet two data fields are present:

- **data**: an array of WORD_LEN bits that represents the data transmitted or received;
- **tx_nrx**: a single bit used to define the master unit mode, a 1 represents tx mode while 0 stands for rx mode;

Both data fields are declared as random using the rand keyword. This aspect is extremely useful for coverage purposes, allowing to generate random values without the direct control of the programmer.

4.10.7 FIFO sequencer

The FIFO sequencer extends the uvm_sequencer and it has the fifo_packet as sequence_item of the class. The function new calls super.new(name, parent) in order to correctly manage the object construction, to pass the arguments to the UVM library. Those arguments can be recalled using methods like get_full_name().

Listing 4.24: fifo_sequencer | new function

```
1 class fifo_sequencer extends uvm_sequencer #(fifo_packet);
2
3   `uvm_component_utils(fifo_sequencer)
4
5   function new (string name, uvm_component parent);
6     super.new(name, parent);
7   endfunction : new
8
9 endclass : fifo_sequencer
```

4.10.8 FIFO sequence

FIFO sequences are defined in the fifo_seq.sv file.
This file is a library of sequences with the fifo_base_seq as the base sequence from where all the other sequences are derived. All the sequences use the fifo_packet as sequence_item. In the pre_body and post_body the objection mechanism is managed, raising the objection in the pre_body and dropping it once the sequence is concluded. From the base_sequence five sequences are derived:

- fifo_write_seq
- fifo_write_over_seq
- fifo_last_seq
- fifo_read_seq

- fifo_read_over_seq

Fifo_write_seq is the sequence dedicated to a simple transmission (TX), instead fifo_write_over_seq manages the scenario in which an overwrite is generated in the FIFO. fifo_read_seq is used in case of a simple reception (RX) and fifo_read_over_seq manages a reception in which a situation of underrun is generated. Lastly the fifo_last_seq is the sequence created for coverage purposes that aims at cover specifically the values not produced randomly. **??** shows the fifo_write_over_seq.

Listing 4.25: fifo_sequence | fifo_write_over_seq

```
//overrun in fifo ==> first write run accordingly but after fifo.
    length fifo.overrun = 1 and no more write
class fifo_write_over_seq extends fifo_base_seq;

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

    `uvm_object_utils(fifo_write_over_seq)

    //BODY
    virtual task body();
        for ( int i = 0; i < `FIFO_MEM_SIZE * 3; i++) begin
            `uvm_info(get_type_name(), $sformatf("Executing sequence
    fifo_write %0d", i), UVM_LOW)
            `uvm_do_with(req, {req.tx_nrx == 1'b1; } )
        end
    endtask : body

endclass : fifo_write_over_seq
```

4.11 I2S

This section is dedicated to the analysis of the I2S protocol related files.

4.11.1 I2S package

The i2s_package is contained in a file called i2s_pkg.sv. This has been created in order to store all the include necessary for a correct implementation of the I2S agent architecture. Due to compilation necessities, files are included using a bottom-up approach, from the sequencer and sequence till the agent.
Additionally, in order to facilitate the use of virtual interfaces in the included files, a new type is declared using the typedef keyword.

Listing 4.26: i2s_pkg | typedef and include

```
    typedef uvm_config_db#(virtual i2s_if) i2s_vif_config;

    `include "config_i2s.sv"
    `include "i2s_packet.sv"

    `include "i2s_monitor.sv"
    `include "i2s_sequencer.sv"
    `include "i2s_seq.sv"
    `include "i2s_driver.sv"

    `include "i2s_agent.sv"
```

4.11.2 I2S agent

The I2S agent is contained in the i2s_agent.sv file. Since I2S must be an active agent, the monitor, driver and sequencer objects are instantiated inside the agent. To guarantee code flexibility, a check on the i2s_active variable is set. In order to allow that, first a check on the configuration object is performed, then, if the i2s_active variable has an active value, a sequencer and a driver are created.

Listing 4.27: i2s_agent | active_passive check and components create

```
    typedef uvm_config_db#(virtual i2s_if) i2s_vif_config;
//get config_obj
        if (!`obj_config(config_i2s)::get(this, get_full_name(), "
    config_i2s", i2s_cfg))
            `uvm_error("NO_CFG", {"configuration object is not set
    properly for: ", get_full_name()})

        mon_i2s = i2s_monitor::type_id::create("mon_i2s", this);
        if(i2s_cfg.i2s_active == UVM_ACTIVE) begin
            seqr_i2s = i2s_sequencer::type_id::create("seqr_i2s",
    this);
            drv_i2s = i2s_driver::type_id::create("drv_i2s", this);
        end
```

Moreover, the i2s_agent shows an example of the management of the ch_id variable in order to distinguish the objects of a channel with respect to the other ones. The ch_id variable is declared as an integer in the agent, its value its taken from the uvm database and set again for the lower levels of the architecture, as it can be seen in listing 4.28

Listing 4.28: i2s_agent | ch_id variable management

```
        //get ch_id from env
        if (!uvm_config_db#(int)::get(this, get_full_name(), "ch_id",
     ch_id))
```

```
3             `uvm_error("NO CH_ID", {"ch_id is not set properly for: "
    , get_full_name()})
4
5         //set ch_id into all lower level components
6         uvm_config_db#(int)::set(this, "*", "ch_id", ch_id);
```

Then, in the connect_phase, there is the connection of the driver to the sequencer constructed using a seq_item_port. This has been implemented to permit the drive of data item generated by the sequencer.

Listing 4.29: i2s_agent | connect_phase

```
1         if(i2s_cfg.i2s_active == UVM_ACTIVE) begin
2             //Connect driver and sequencer
3             drv_i2s.seq_item_port.connect(seqr_i2s.seq_item_export);
4         end
```

4.11.3 I2S driver

The i2s_driver is used to send data to the DUT in the correct time frame following the specifics of the protocol. In the run_phase, the first action is setting the sd_reg signal to Z, i.e., high impedance. This is done in order to avoid conflicts on the line, setting it to high impedance.
Few wait conditions are used in order to synchronize the driver with the DUT, then a forever loop is used to start the sending of data. Line 9 of listing 4.30 is used to control that only the i2s_agent associated with the correct channel performs the operation. Then, according to the value of tx_nrx signal of the interface, the correspondent task is selected. In case of a transmission the send_data task is executed, else the DUT is in reception mode and the get_data task is performed.

Listing 4.30: i2s_driver | run_phase

```
1         vif_i2s.sd_reg = 1'bz;
2
3         wait(vif_i2s.enable);
4         wait(vif_i2s.rst_l);
5         @(negedge vif_i2s.sck)
6         wait(vif_i2s.ws == 0);
7
8         forever begin
9             wait(vif_i2s.ws == ch_id);                    //if
    this is the correct channel
10            if (!vif_i2s.tx_nrx)
11                send_data();
12            else
13                get_data();
14        end
```

The send_data task is responsible for the sending of transactions generated by the i2s_sequencer.
Using the seq_item_port.get_next_item(req) method, a data item is taken from the sequencer. The for cycle at line 3 is used in order to save SD on the array of WORD_LEN bits one bit at each clock cycle on the negative edge of the clock.

Listing 4.31: i2s_driver| run_phase - send_data

```
1         seq_item_port.get_next_item(req);
2
3         for(int i = `WORD_LEN - 1; i >= 0 ; i--) begin
4             @(negedge vif_i2s.sck)
5             vif_i2s.sd_reg = req.sd[i];
6         end
7
8         @(posedge vif_i2s.sck) void'(this.begin_tr(req, "I2S
    Transaction"));
9         `uvm_info(get_type_name(), $sformatf("Driving transaction :\n
    %s",req.sprint()), UVM_MEDIUM)
10        this.end_tr(req);
11
12        // Communicate item done to the sequencer
13        seq_item_port.item_done();
14
15    endtask : send_data
```

The opposite purpose is managed by the get_data task.
A i2s_packet instance called data_collected is created in order to store the data received. A for cycle is used to manage the reception of all data, one bit per clock cycle, followed by the correct assignment of the tx_nrx value.

Listing 4.32: i2s_driver | run_phase - get_data

```
1     i2s_packet data_collected;
2
3     @(negedge vif_i2s.sck)
4     seq_item_port.get_next_item(req);
5
6     data_collected = i2s_packet::type_id::create("data_collected",
    this);
7
8     for(int i = `WORD_LEN - 1; i >= 0 ; i--) begin
9         @(negedge vif_i2s.sck)
10        data_collected.sd[i] = vif_i2s.sd;
11    end
12
13    data_collected.tx_nrx = vif_i2s.tx_nrx;
14
```

```
15      @(posedge vif_i2s.sck) void'(this.begin_tr(req));
16      this.end_tr(req);
17      seq_item_port.item_done();
```

4.11.4 I2S interface

The interface is called i2s_if. This file contains all the signals needed to connect the i2s_agent to the DUT.
Among all signals, it is important to highlight the SD signal, declared as a wire, and a logic bit WS. The reason why the SD signal is implemented as a wire and it is managed by means of *sd_reg*, that is, a backing variable, stems from the fact that it is used both as input and output. The assignment at line 7 listing 4.33 is used to simulate the behaviour of the wire in the pad block of the I2S protocol. Lastly, lines 9 to 11 of listing 4.33 are used for coverage purposes, the property keyword represents an Assertion used to check the occurrence of some specific scenarios between the listed signals. The use of assertions is further explained in chapter 5.

Listing 4.33: i2s_if| signals and assertion

```
1     //from dut to if
2     wire sd;
3     logic ws;
4
5     logic sd_reg;
6
7     assign sd = sd_reg;
8
9     property ws_down_stable;
10        @(posedge sck) $fell(ws) |-> ($past(ws, 15) == 1'b1);
11    endproperty
12
13    ws_down_p: assert property (ws_down_stable);
```

4.11.5 I2S monitor

The i2s_monitor is a passive component that is used to keep track of all the transmitted and received data.
The connection_phase is used to control the correct instantiation of the virtual interface (line 3 of listing 4.34) and the presence of a configuration object (line 5 of listing 4.34), other than the channel id. In case of an error, a uvm_error is displayed, reporting the error type too.

Listing 4.34: i2s_monitor | connect_phase

```
1         // get virtual interface
```

```
2         if (!i2s_vif_config::get(this, get_full_name(),"vif_i2s",
      vif_i2s))
3             `uvm_error("NOVIF",{"virtual interface must be set for: "
      ,get_full_name(),".vif_i2s"})
4         //get config_obj
5         if (!`obj_config(config_i2s)::get(this, get_full_name(), "
      config_i2s", i2s_cfg))
6             `uvm_error("NO_CONFIG", {"configuration object is not set
       properly for: ", get_full_name()})
7         else
8             `uvm_info(get_full_name(), "ok get::cfg", UVM_HIGH)
                                                        //debug
9
10        //get ch_id
11        if(!uvm_config_db#(int)::get(this, get_full_name(), "ch_id",
      ch_id))
12            `uvm_error("NO_CH",{"ch_id must be set for: ",
      get_full_name()})
```

In the run_phase, instead, the storing of the received data is performed. A i2s_packet, called data_collected, is declared and used to save the bits sent or received at each negative edge of the clock. Once that a i2s_packet is completely stored, tx_nrx included, the i2s_packet is sent to the scoreboard using the TLM port.
In the Scoreboard, the data item is compared to the one obtained in the FIFO side to check for the correctness of the transmission.

Listing 4.35: i2s_monitor | run_phase

```
1         i2s_packet data_collected;
2
3         vif_i2s.sd_reg = 1'bz;
4
5         wait(vif_i2s.enable);
6             wait(vif_i2s.rst_l);
7             @(negedge vif_i2s.sck)
8             wait(vif_i2s.ws == 0);
9
10            forever begin
11                wait(vif_i2s.ws == ch_id);
12                @(negedge vif_i2s.sck)
13                //create sequence_item
14                data_collected = i2s_packet::type_id::create("
      data_collected", this);
15
16                for(int i = `WORD_LEN - 1; i >= 0; i--) begin
17                    @(negedge vif_i2s.sck)
18                    data_collected.sd[i] = vif_i2s.sd;
19                end
```

```
20
21              //send packet to Scoreboard using TLM port
22                  i2s_write_port.write(data_collected);
23              end
```

4.11.6 I2S packet

I2S packet is the data item used by the architecture of the i2s_agent, setting the data fields that will be filled by the data items generated at each transaction request.
In the i2s_packet, two data fields are present:

- sd : serial data, it represents the actual bits of data transmitted or received,
- tx_nrx : control bit used to identify the working mode. It is set to 0 in case of RX, while 1 is used for a TX.

Listing 4.36: i2s_packet | data fields

```
1     rand bit [ `WORD_LEN - 1 : 0 ]    sd;
2     rand bit                          tx_nrx;
```

Both data fields are declared with the keyword "rand". Thanks to this keyword, once a data item is created the two data fields are generated randomly with a uniform distribution.
The sd data field is an array of WORD_LEN bits, it hasn't been set as a simple bit like the actual SD of the I2S protocol controls in order to more smoothly compare input and output data of the I2S protocol during the coverage phase.

4.11.7 I2S sequencer

The i2s_sequencer is fundamental for data transmission to the driver, that is in charge of driving them to the interface. It is defined as an extension of the uvm_sequencer and uses the i2s_packet as a data item.

Listing 4.37: i2s_sequencer | class

```
1
2  class i2s_sequencer extends uvm_sequencer #(i2s_packet);
3
4    `uvm_component_utils(i2s_sequencer)
5
6    function new (string name, uvm_component parent);
7      super.new(name, parent);
8    endfunction : new
9
10 endclass : i2s_sequencer
```

4.11.8 I2S sequence

The i2s_seq.sv is the file containing the sequences used by the i2s_sequencer. All sequences created have been derived from a common one: i2s_base_seq, that uses the i2s_packet aforementioned. All the sequences are controlled by the objection mechanism in order to lock the system in the run_phase for all the required time. Five sequences are created:

- i2s_read_seq
- i2s_read_over_seq
- i2s_write_seq
- i2s_write_over_seq
- i2s_last_seq

Since the I2S side has to work directly with the FIFO side, the sequences are very similar. It is important to notice that each write on the I2S side is associated with a read on the FIFO side and vice-versa.

The read sequences differ on the number of data items received on the protocol, with the i2s_read_seq that is used to receive FIFO_MEM_SIZE - 5 data items and i2s_read_over_seq that expects FIFO_MEM_SIZE * 3 data items.
For what concerns the write sequences, the only difference is that these sequences are used to send data items in the protocol, setting the tx_nrx bit to 1 to correctly set the working mode. In listing 4.38, the i2s_read_over_seq is shown.

Listing 4.38: i2s_sequence | i2s_read_over_seq

```
1  class i2s_read_over_seq extends i2s_base_seq;
2
3      function new(string name = "i2s_read_over_seq");
4          super.new(name);
5      endfunction : new
6
7      `uvm_object_utils(i2s_read_over_seq)
8
9      //BODY
10     virtual task body();
11         for (int i = 0; i < `FIFO_MEM_SIZE * 3; i++) begin
12             `uvm_info(get_type_name(), $sformatf("Executing sequence
       i2s_read %0d", i), UVM_LOW)                          //debug
13             `uvm_do_with(req, { req.sd == 0;
14                                 req.tx_nrx == 1'b1;})
15         end
16     endtask : body
```

```
17
18 endclass :i2s_read_over_seq
```

4.12 CFG

The CGF unit is the simplest one. It is the UVC dedicated to the management of the control signals such as the enable, reset and the tx_nrx signals.

4.12.1 CFG package

The package is contained in the cfg_pkg.sv file. In this file all the include related to the CFG are contained other than the type creation of the virtual interface.

Listing 4.39: cfg_package | include and typedef

```
1
2     typedef uvm_config_db#(virtual cfg_if) cfg_vif_config;
3
4     `include "cfg_packet.sv"
5
6     `include "cfg_monitor.sv"
7     `include "cfg_sequencer.sv"
8     `include "cfg_seq.sv"
9     `include "cfg_driver.sv"
10
11    `include "cfg_agent.sv"
```

4.12.2 CFG agent

The cfg_agent is saved in the cfg_agent.sv file. It manages the presence of the submodules in the build_phase using a check on the value of the is_active variable. In case the variable is equal to UVM_ACTIVE, the architecture is composed by a monitor, a sequencer and a driver. On the other hand, if the agent is passive it only contains the monitor.

Listing 4.40: cfg_agent | build_phase

```
1     super.build_phase(phase);
2         mon_cfg = cfg_monitor::type_id::create("mon_cfg", this);
3         if(is_active == UVM_ACTIVE) begin
4             seqr_cfg = cfg_sequencer::type_id::create("seqr_cfg",
    this);
5             drv_cfg = cfg_driver::type_id::create("drv_cfg", this);
```

```
6         end
```

In the connect_phase, instead, the connection between the driver and the sequencer is created, using the connect method.

Listing 4.41: cfg_agent | connect_phase

```
1     if(is_active == UVM_ACTIVE) begin
2             //Connect driver and seqr
3             drv_cfg.seq_item_port.connect(seqr_cfg.seq_item_export);
4     end
```

4.12.3 CFG driver

The cfg_driver, as usual, has configuration checks in the build_phase to ensure the correctness of the architecture construction. The run_phase, on the other hand, first initializes the three controlled signals to 0.
Then, a forever loop guarantees that the driver works as long as needed: at each positive edge of the clock a sequence item is transmitted controlling the values of enable, tx_nrx (signal controlling the working mode) and the reset.

Listing 4.42: cfg_driver | run_phase

```
1
2             vif_cfg.tx_nrx = 0;
3             vif_cfg.enable = 0;
4             vif_cfg.rst_l  = 0;
5
6         forever begin
7             @(posedge vif_cfg.clk)
8             seq_item_port.get_next_item(req);
9             void'(this.begin_tr(req, "CFG Driver Transaction"));
10            `uvm_info(get_type_name(), "Driving enable and mode",
    UVM_LOW)
11            vif_cfg.enable = req.enable;
12            vif_cfg.tx_nrx = req.tx_nrx;
13            vif_cfg.rst_l = req.rst_l;
14            this.end_tr(req);
15            seq_item_port.item_done();
16        end
```

4.12.4 CFG interface

All signals controlled by the CFG UVC are signals that are declared in the top entity, without adding signals used specifically for this UVC. For this reason, the interface is declared with all the controlled signals in the sensitivity list, as shown in listing 4.43 .

Listing 4.43: cfg_interface | class

```
interface cfg_if(input clk, output logic rst_l, output logic enable,
    output logic tx_nrx);


endinterface : cfg_if
```

4.12.5 CFG monitor

The monitor in this UVC is primarily used for debug purposes, since it does not receive data and inject not random signals value.
In the run_phase, the phase is controlled by a forever loop with a wait condition used to synchronize the class with the rest of the architecture. At each positive edge of the clock, a data item is created and transmitted to the DUT.

Listing 4.44: cfg_monitor | run_phase

```
    forever begin
            wait(!vif_cfg.rst_l)
            @(posedge vif_cfg.clk)
            cfg_pkt = cfg_packet::type_id::create("cfg_pkt", this);
            void'(this.begin_tr(cfg_pkt, "CFG Monitor Transaction"));
            this.end_tr(cfg_pkt);
        end
```

4.12.6 CFG packet

Due to its limited functionalities the data item's structure is very simple, given the fact that there is no necessity of sending data but only setting some signals.

4.12.7 CFG sequence

In the cfg_seq.sv file, five sequences related to the CFG UVC are contained. Due to the UVC characteristics, the sequences set the appropriate values of the control signals. The five sequences are:

- cfg_write_seq
- cfg_read_seq
- cfg_drop_enable_rst_seq
- cfg_raise_enable_rst_seq

The cfg_write_seq is the sequence that raises the enable signal, sets the tx_nrx signal at 1 meaning a TX working mode and deactivates the reset setting it to 1.
The cfg_read_seq is the opposite sequence with respect to the aforementioned one, as it sets the tx_nrx signal to 0.
The cfg_drop_enable_rst_seq is used to drop the enable signal and also set the reset to 0.
Lastly, the cfg_raise_enable_rst_seq is used to raise the enable signal and deactivate the reset one.

4.12.8 CFG sequencer

The cfg_sequencer is the component used to send the generated data_item to the driver. It uses the cfg_packet as data_item.

Listing 4.45: cfg_sequencer | class

```
1 class cfg_sequencer extends uvm_sequencer #(cfg_packet);
2
3   `uvm_component_utils(cfg_sequencer)
4
5   function new (string name, uvm_component parent);
6     super.new(name, parent);
7   endfunction : new
8
9 endclass : cfg_sequencer
```

Chapter 5

Coverage, Assertions, Regressions, Merging, Results

The final phase of the MDV cycle is dedicated to the application of the proposed tests and the analysis of the obtained results.
In order to understand if the code developed for this thesis project covers all the possible scenarios and all the requested features, coverage data is essential.
Coverage is the measure of how much the DUT has been tested according to the test plan, providing us insights about the system's controllability and observability. Controllability is the ability of triggering a Finite State Machine (FSM) state, a specific line of code or a working mode by stimulating a subset of input ports, only. Observability, on the other hand, is related to the ability of monitoring the effects of a specific FSM state, code line or working mode.[15]
In conclusion, coverage data gives us information about what portion of code has been activated by the tests and if all the planned features have been adequately tested, in addition to what results have been produced.
Two types of *Coverage* can be primarily identified:

- Code coverage
- Functional coverage

5.1 Code Coverage

Code coverage is undoubtedly the most straightforward and direct type of coverage as it doesn't need any change in the code or code lines written for this specific

purpose.
Code Coverage gives the verification engineer information on what line or structure of the code has been tested. It is categorized into the following types [15] [16]:

- **Toggle Coverage**: measures if and how many times the value of each bit of a register, or wire, has toggled,
- **Line Coverage**: measures how many times each code's line has been executed. It is not unusual that a minimum number of times higher than 1 is set to obtain a pass,
- **Statement Coverage**: evaluates which statement in the code has been tested highlighting which part of each statement has not been executed,
- **Block Coverage**: detects if a block of code has been tested,
- **Branch Coverage**: identifies if both the possible branches, namely true or false, of a branch statement have been tested,
- **Expression Coverage**: measures if a Boolean expression has been tested both on true and false input,
- **Finite State Machine Coverage**: controls if all the available states have been checked and tested.

5.2 Functional Coverage

Code Coverage is very useful, although on its own it is not capable of providing sufficient information to determine if the DUT has been properly tested.
With the introduction of constrained-random stimuli in simulations, functional coverage allows to generate accurate reports showing which inputs have been used in the tests and which have not. Due to the random nature of the inputs, precisely determining what input data has been used without manually inspecting the output waveforms is a challenging task for the verification engineer. In addition to that, such inspection often proves to be a time-consuming and tedious process. In order to save time and make the job easier, reports have been generated by leveraging the features of functional coverage, producing a set of organized and straightforward results.
Unfortunately, this coverage type is not automatic and requires code modifications in order to be implemented.

5.2.1 Cover Group, Cover Points and Cross Coverage

To obtain an accurate functional coverage there is the necessity of testing all the signals of the DUT and monitor which stimuli has been used.

Cover groups and *Cover points* are two user-defined metric used to specify a coverage model and to provide an accurate control on which inputs have been randomly used.
A cover group is an object-type used to monitor variable values and it is defined using the cover group keyword. Within a cover group, cover points are declared in order to specify which values have to be tracked.
Given an input of the DUT, a cover point can be a specific value or a range of values that must be covered during simulations to obtain a pass. Additionally, a minimum number of occurrences can be defined for each cover point, indicating how many times a specific value must occur to consider the test successful.
Unfortunately, the application of a value in a specific input is not always sufficient to generate a certain output or trigger a particular event, as it is possible that more than one input must have a specific value at the same time. In this case the use of *Cross Coverage* is essential. Cross coverage controls the cross-product between two cover points previously defined. In other words, it checks that all the combinations of the two cover points are generated. Only when all the possible combinations have been evaluated is the cross coverage considered completed and passed [16].

5.3 Assertion Based Verification

In addition to the functional coverage, the Assertion Based Verification (ABV) is used to verify the design correctness.
This type of verification combines characteristics of the formal and functional verification using Assertions as the main structure. An assertion is a SystemVerilog feature that expresses a condition that must always be true, ensuring that the design behaves as expected under specified conditions [17]. Two types of assertions can be distinguished:

- Immediate Assertions
- Concurrent Assertions

5.3.1 Immediate Assertions

An immediate assertion is used to check a condition at the current simulation time, specifying which actions have to be performed in case of a success or a failure. Assertions are construct used to verify the correctness of a statement, with failures typically associated with a certain severity level. By default the severity is set at

'error'.
This structure is identified by the keyword: 'assert' followed by the expression that has to be checked [18].

5.3.2 Concurrent Assertion

A concurrent assertion controls a sequence of events over one or multiple clock cycles resulting in a continuous check throughout the simulation. To differentiate the two types of assertions, they is identified by the words 'assert property', followed by the events to be tracked [18].

5.4 Testbench Coverage

This thesis' aim is to verify all features and code lines of the DUT. Achieving this goal requires the execution of a large number of tests to cover every obtainable result through the injection of all potential input combinations.
In the developed testbench the features identified in the Plan phase have been tested using the seven different tests described in the Test section. A single run is typically not enough to generate a sufficient coverage. For example the tx_over_seq in the tx_over test injects only 20 random stimuli for each run. It is, therefore, clear why it is necessary to compound the results of multiple runs in order to obtain a full coverage.

In order to launch a test, multiple procedures can be followed. The goal is to automate the majority of the work as well as optimizing and minimizing the time spent manually writing code lines in the terminal.

The simulation is executed using Cadence Xcelium Logic Simulator. The simulator uses a file called run.f that contains all the information needed to compile and execute the testbench environment developed. To get more into details, it has all the *include* statements related to the directives containing files that are used in the testbench, as shown in listing 5.1.

Listing 5.1: run.f file | include directories

```
1 //include directories
2 -incdir ../env
3 -incdir ../tests
4 -incdir ../bfm/fifo
5 -incdir ../bfm/i2s
6 -incdir ../bfm/cfg
7 -incdir ../cov
```

Moreover, this file contains specifications of the access permissions given to all the objects in the design during the simulation. Possible permissions are: read, write and connectivity (line 1 of listing 5.2). While read and write are straightforward, connectivity is chosen to provide information useful for signals traceability across the DUT hierarchy.
Then, two input specifications are used to include two TCL files: waves.tcl and run.tcl (line 2-3 listing 5.2). The first line provides important information for the representation and saving of the output waveforms, while run.tcl simply includes the run command.

Listing 5.2: run.f file | permissions and input files

```
1 -access +rwc
2 -input waves.tcl
3 -input run.tcl
```

Following that, a few lines are used to specify characteristics of the run:

- UVM_TESTNAME: provides the name of the test to be executed. The name has to coincide with the one declared in the test class,
- UVM_VERBOSITY: expresses the verbosity level, that is, the amount of information and debug messages contained in the output terminal. Six levels of verbosity exist: NONE, LOW, MEDIUM, HIGH, FULL and DEBUG. By default it is set at MEDIUM.
  All the messages with a verbosity lower or equal to the level set will be displayed,
- SVSEED: is an option used to set the seed of the value generated in the testbench.

Listing 5.3 shows an example of how the options have been set in a run. The selected test is tx_over_test, with HIGH verbosity and a random seed. Due to the limited number of data combinations the SVSEED has been left as random for every run executed, reaching whatsoever the total coverage in a very reasonable time. In a different situation with bigger data sets, it could be appropriate to change coherently the seed to obtain some specific values with a higher frequency.

Listing 5.3: run.f file | test options

```
1 //test options
2 +UVM_TESTNAME = tx_over_test
3 +UVM_VERBOSITY = UVM_HIGH
4 +SVSEED = random
```

After the run options, the files that need to be compiled for the testbench are listed with a bottom to top order, with final entry being the file containing the

RTL of the DUT. Listing 5.4 shows the final segment of run.f with files divided by testbench unit.

Listing 5.4: run.f file | testbench files and rtl

```
../env/define_pkg.sv

../bfm/fifo/fifo_pkg.sv
../bfm/fifo/fifo_if.sv

../bfm/cfg/cfg_pkg.sv
../bfm/cfg/cfg_if.sv

../bfm/i2s/i2s_pkg.sv
../bfm/i2s/i2s_if.sv

//top module for uvm_module
../env/top.sv

//rtl
../../i2s_block/rtl/i2s_master_pkg.sv
../../i2s_block/rtl/i2s_dp_fifo.sv
../../i2s_block/rtl/i2s_pads.sv
../../i2s_block/rtl/i2s_protocol.sv
../../i2s_block/rtl/i2s_master_txrx.sv
```

Before introducing the concept of regression tests and discussing the related results, it is useful to deeply analyze the last not described file of the testbench implemented: coverage.sv.

5.4.1 Coverage.sv

Coverage file is the file in charge of defining the cover groups and cover points useful to obtain an accurate coverage of the DUT. It is defined as a uvm_component extension. The file can be divided in 4 main parts:

- new function: mandatory to construct the component in the testbench;
- build_phase
- i2s_cg covergroup
- fifo_cg covergroup

In the build_phase, the correct declaration of the interface is controlled for both the FIFO and I2S agents. The construction is quite simple: an if structure checks the presence of the virtual interface, and in case it is not found, it displays an uvm_error. The presence of the virtual interface is checked using the get method of the uvm_database.

Listing 5.5: coverage.sv file | build_phase

```
1
2 if (!i2s_vif_config::get(this, get_full_name(),"vif_i2s_cov", vif_i2s
  ))
3     `uvm_error("NOVIF",{"virtual interface must be set for: ",
  get_full_name(),".vif_i2s_cov"})
4
5 if (!fifo_vif_config::get(this, "","vif_fifo_cov", vif_fifo))
6     `uvm_error("NOVIF",{"virtual interface must be set for: ",
  get_full_name(),".vif_fifo_cov"})
```

On the other hand, for what concerns the coverage of the FIFO agent some cover points have been created.
The adopted approach consisted in setting a different cover point for each signal in the fifo_interface so: data_tx, data_tx_put_en, data_rx, data_rx_get_en, fifo_overrun and fifo_underrun. In listing 5.6, the two possible types of cover point declarations can be observed. For data_tx_put_en and data_rx_get_en the bins have been explicitly declared, specifying the exact grouping of the values and allowing for better organized results and more meaningful reports. In this case, since these are two single bit signals the division is very straightforward.
Different case is the one concerning data_tx and data_rx, as no bin has been specified for those two signals. The absence of an explicit bins declaration results in an automatic division of the total range of values of the cover point into 64 identical intervals. Due to the data length, i.e., 16 bits, and the uniform probability of the single values, the division into 64 bins is considered acceptable for a good coverage of the whole range.
If the specifications change and a different grouping of values is needed, those ranges can be modified as it is more convenient. An example is shown in listing 5.7.
Analyzing in depth the listing 5.6: at line 2 there is the " option.per_instance = 1 ". This method is used to fix the minimum number of times each bin has to be triggered to be considered as tested. In the example it is set to 1; this is a quite conservative choice aimed to have a low total compilation time.

Listing 5.6: coverage.sv file | fifo_cg covergroup

```
1 covergroup fifo_cg @(posedge vif_fifo.clk);
2         option.per_instance = 1;
3
4         data_tx:         coverpoint vif_fifo.data_tx;
5
6         data_tx_put_en: coverpoint vif_fifo.data_tx_put_en
7                     { bins zero ={0};
8                       bins one  ={1}; }
9         data_rx :        coverpoint vif_fifo.data_rx;
10        data_rx_get_en: coverpoint vif_fifo.data_rx_get_en
11                    { bins zero ={0};
```

```
12                      bins one  ={1}; }
13          fifo_overrun:   coverpoint vif_fifo.fifo_overrun;
14          fifo_underrun:  coverpoint vif_fifo.fifo_underrun;
15
16          put_enXtx:      cross data_tx_put_en, data_tx { ignore_bins
    put_en_zero = binsof(data_tx_put_en.zero);}
17          get_enXrx:      cross data_rx_get_en, data_rx { ignore_bins
    get_en_zero = binsof(data_rx_get_en.zero);}
18
19  endgroup : fifo_cg
```

Listing 5.7: coverage.sv file | data_tx coverpoint

```
1   data_tx:        coverpoint vif_fifo.data_tx;
2           {
3               bins zero           = {0};
4                 bins low_val  = {[1 : 16384]};
5               bins mid_val  [64]  = {[16384 : 32768]};
6               bins high_val [64] = {[32769 : 65535]};
7               bins max            = {65535};
8           }
```

In the last few lines of fifo_cg, lines 16 and 17 of listing 5.6, an example of cross coverage can be observed. The cross coverage is identified by the keyword cross before the two types of cover points involved, after that some additional commands can be inserted between curly braces. At line 16, the the ignore_bins statement is included, with the goal of telling the simulator not to evaluate a precise combination of the cross coverage. In the reported scenario, the ignored bin is the value 0 of data_tx_put_en.

Due to the agent simplicity with respect to the FIFO one, the i2s_cg is much simpler than fifo_cg. Even if there are no significant differences, for the sake of completeness, i2s_cg is reported in the listing 5.8.

Listing 5.8: coverage.sv file | fifo_cg cross coverage

```
1   covergroup i2s_cg @(posedge vif_i2s.sck);
2
3           option.per_instance = 1;
4
5           sd:             coverpoint vif_i2s.sd;
6           ws:             coverpoint vif_i2s.ws;
7
8           sdXws:          cross sd, ws;
9
10  endgroup : i2s_cg
```

Once all the required cover group and cover points have been defined, the test can be run.

Due to the necessity of running multiple times the simulation the usual solution is the construction of a script that automatize the process. This script is called *Regression*, it has to execute the run multiple times with various configurations, test and, if , changing the seed for randomization.
After selecting a reasonable amount of runs to obtain meaningful results and having completed the regression run, the reports can be analyzed on IMC.
IMC is a powerful software developed by Cadence that allows to organize and show the results of a run or a regression in an ordinate and intuitive report, with a visual representation of all the analyzed coverage types and their associated covered percentage.
In order to correctly analyze the results it is important to remember some information:

- FIFO_MEM_SIZE is set to 8,
- the sequence fifo_write_seq of the FIFO UVC generates a number of data items to be transmitted equal to FIFO_MEM_SIZE - 5, so 3 data items,
- the sequence fifo_write_over_seq of the FIFO UVC generates a number of data item to be transmitted equal to FIFO_MEM_SIZE * 3, so 24 data items,
- the I2S sequences replicates the functioning of the FIFO ones in terms of number of generated data items.

Taking into account all these information, it was decided to execute the run.f file 10 times. Remembering the aforementioned data, it has been estimated that the generation of 240 random stimuli, throughout the ten runs, would be a reasonable number of inputs to have a very high probability of covering all the bins created without increasing too much the total simulation time. Special care has been placed in making sure that the data_tx field, with its 64 bins, is thoroughly verified and no bin is left unchecked.
The regression script 'regression.sch' contains a simple for cycle that executes the run.f file 10 times. The run.f file is executed using the xrun command adding some options in order to obtain the coverage results and being able to analyze them on IMC.
The added options are:

- **-coverage**: this option enables coverage data generation. The coverage type can be specified using a letter after the keyword: B for Block Coverage, E for expression, F for FSM, T for toggle, U for functional and A for all the types,
- **-covscope**: used to specify the scope in which the results will be stored,
- **-covtest**: used to specify the name of the test executed during this run,

- **-covoverwrite**: used to enable the overwriting of coverage output files and directories.

Listing 5.9: regression.sch file

```
1  #!/bin/sh
2
3  for i in $(seq 0 9);
4  do
5      echo $i
6      xrun -f run.f -coverage A -covscope tx_test -covtest tx_$i
       -covoverwrite
7  done
8
9  imc -exec merge.cmd
10 imc -load cov_work/rx_test/rx_merge_all
```

In the last lines of the regression file are reported some IMC command lines. Line 9 and 10 of listing 5.9 are used for merging purposes.
Merging is a methodology employed to aggregate the various data obtained from each single run together in order to obtain a comprehensive view of what has been covered.
IMC doesn't support a graphic interface to perform the merge, but it requires the use of the batch mode. The batch mode needs the commands insertion through the use of terminal or through scripts in order to be performed. If needed, the merge action is often automatized inside the regression script to save time at each new simulation. In order to avoid the manual insertion of code lines on the terminal, line 9 is used. The -exec option guarantees the possibility to launch a script and then close IMC. In this thesis work, such script is named merge.cmd. As it can be seen in listing 5.10, a single line of code is contained in merge.cmd and it is essential to obtain the desired merging.

Listing 5.10: merge.cmd file

```
1  merge cov_work/rx_test/* -out cov_work/rx_test/rx_merge_all -
       initial_model union_all -overwrite
```

The command shown above can be divided in few parts:

- path of the files that must be merged. In the reported example, there is only one path, but it ends with a *. This wildcard is used to represent all the files contained in the rx_test folder so that they can be merged together,
- -out option: specifies the path of the directory where the results will be stored,
- -initial_model which specify the target model for the merge operation. It can accept few arguments: -union_all, primary_run and -empty. In the code

shown above, the union_all option has been chosen, ensuring the merge of all data without the exclusion of neither of them;

- -overwrite option, this has been used due to the need of overwriting some directories previously created;

Once all the regression settings have been set and after deciding the test to be run, the regression is launched.
The first regression has been run with the tx_over_test, saving the results in the directory tx_test and merging all the obtained results in the merge_all directory using the -union_all model.
On IMC, the obtained results can be analyzed and grouped as shown in fig. 5.1.

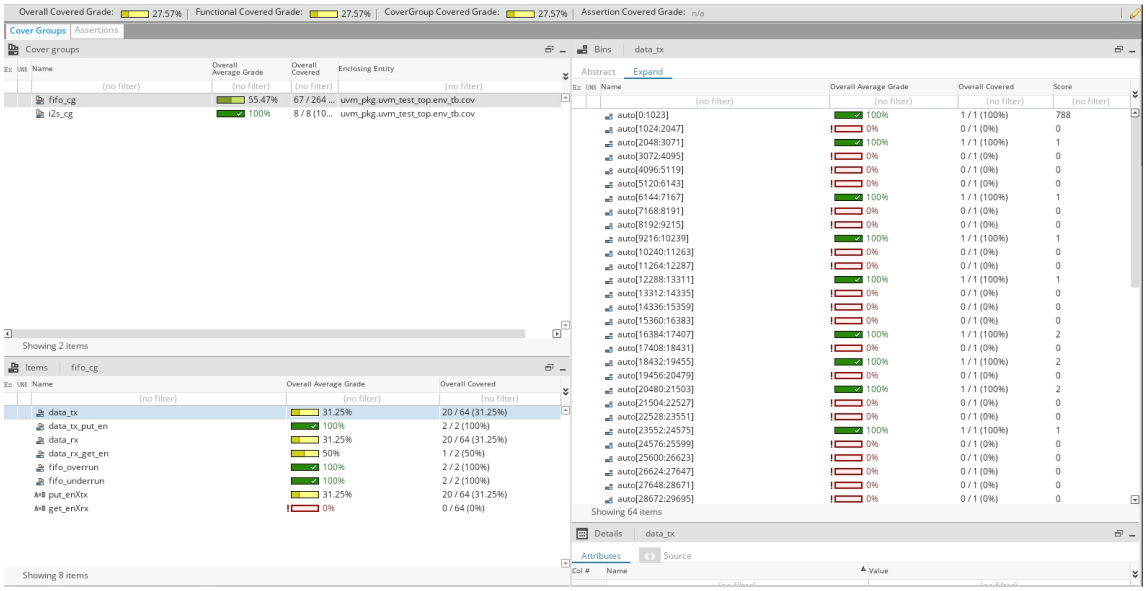

Figure 5.1: First run TX

As it can be seen after a single run, 31.25% of data_tx bins has been covered. The tested bins can be observed in detail in the right column of fig. 5.1.
The obtained coverage is not sufficient: this is given by the limited number of inputs tested,leading to a low percentage of bins covered.
More meaningful results can be analyzed after the regression execution, that is, 10 runs of the test. To better organize and understand the results obtained, the single results were merged together by means of the merge command, thus obtaining the total regression coverage. For what concerns the data_tx field, the percentage of bins covered is higher than the 90% with only two bins not covered, while all the other fields related to the TX transmission have been fully covered.
To ensure the coverage of the remaining bins, the test tx_last_test has been used.

This test uses the sequence fifo_last_seq of the fifo_sequencer that has been specifically created to hit the remaining three bins not covered. The sequence can be observed in listing 5.11.

Listing 5.11: fifo_seq.sv file | fifo_last_seq

```
//sequence created to hit the last three bins not covered into
    coverage
class fifo_last_seq extends fifo_base_seq;

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

    `uvm_object_utils(fifo_last_seq)

    //BODY
    virtual task body();
        for (int i = 0; i < 3; i++) begin
            `uvm_info(get_type_name(), $sformatf("Executing sequence
    fifo_write %0d", i), UVM_LOW)
            if (i == 1)
                `uvm_do_with(req, {req.data == 4000;
                                   req.tx_nrx == 1'b1; } )
            else if (i == 2)
                `uvm_do_with(req, {req.data == 31000;
                                   req.tx_nrx == 1'b1; } )
            else
                `uvm_do_with(req, {req.data == 35000;
                                   req.tx_nrx == 1'b1; } )
        end
    endtask : body

endclass : fifo_last_seq
```

Through the combination of a for loop and the uvm_do_with macro the data field is forced to the desired value, obtaining the three wanted values in three distinct cycles. The three assigned values are chosen arbitrarily by analyzing the IMC report and picking a value for each remaining bin.
The proposed approach is not the only possible one, as the total coverage can also be reached through the execution of a higher number of test runs. Unfortunately, increasing the number of runs has as a direct drawback as it increases the amount of time required to conclude the regression, without guaranteeing the mathematical certainty of a full coverage.
The random driven approach used to fulfill the coverage of the majority of the bins plus the generation of few constrained values to cover the last bins is the best trade-off between time required for simulations and time spent to write additional

code.
After the run of the tx_last test, the obtained results have been merged with the ones of the regression. As shown in fig. 5.2, the coverage percentage has drastically improved.

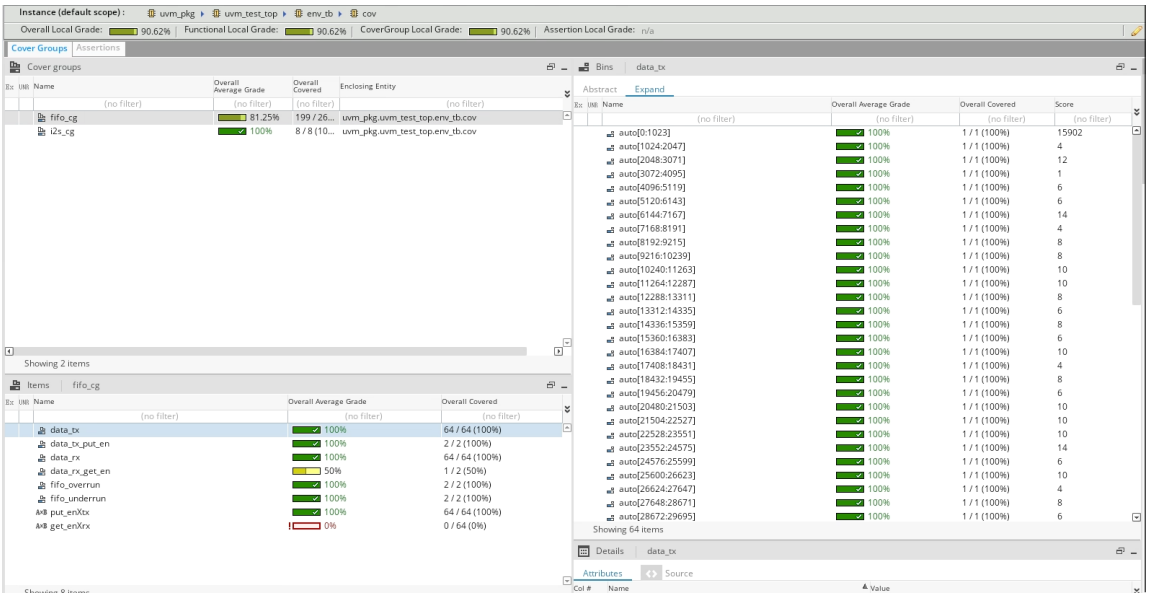

Figure 5.2: TX coverage after merge

Although all bins of data_tx have been hit, the observed coverage remains below 100%.
By analyzing the single cover points, it is possible to see how signal data_get_en has its 0 value not covered. This information, if not contextualized, can be seen as a missed covered value. However, knowing how the i2s protocol works, it is possible to determine that the value 1 of the aforementioned signal can be triggered only during the reception mode when a value is received in the FIFO block. Consequently, it is correct that it has not been covered yet, given the set of tests that have been launched so far.
The analysis of the coverage results and the exclusion of some cover points impossible to be covered is called *refinement*, and it is an important step in the whole coverage process.
The refinement ensures that the coverage percentage really reflects the amount of scenarios covered.
In the coverage report, the refinement process has been done using the *exclusion* feature of IMC, obtaining a fully coverage for what concerns the Transmission mode.

As shown in the fig. 5.3 the value 0 of the data_rx_get_en and the cross coverage get_enXrx, directly affected by the previous signal coverage, have been excluded from the coverage through refinement.

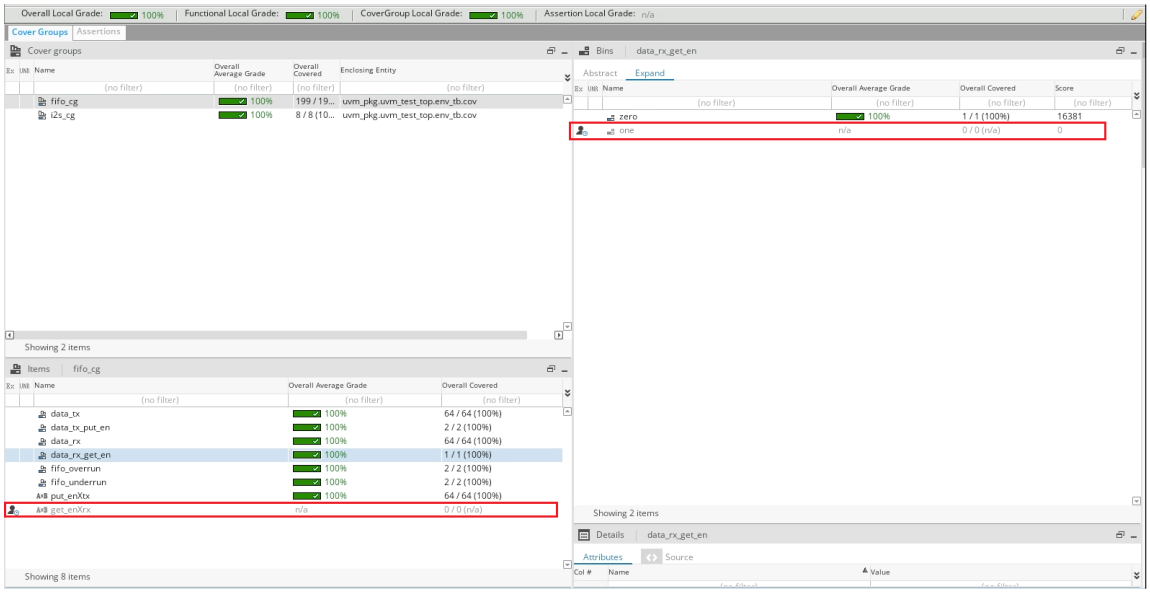

Figure 5.3: TX coverage after merge and refining

Then, the test has been changed with the rx_over_test to also verify the reception. After that first regression, the test has been changed to rx_last in order to also hit the remaining not covered bins.
The sequence run by the i2s_sequencer is rx_write_last and it is reported in listing 5.12.

Listing 5.12: i2s_seq.sv file | i2s_last_seq

```
1  class i2s_last_seq extends i2s_base_seq;
2
3      function new(string name = "i2s_last_seq");
4          super.new(name);
5      endfunction : new
6
7      `uvm_object_utils(i2s_last_seq)
8
9      //BODY
10     virtual task body();
11         `uvm_info(get_type_name(), $sformatf("Executing sequence
       i2s_write"), UVM_LOW)
12
13         `uvm_do_with(req, { req.sd == 25000;
14                             req.tx_nrx == 1'b0;} )
```

```
15      endtask : body
16
17  endclass :i2s_last_seq
```

As for the TX mode, the value of sd in the i2s_last_seq has been chosen observing the results obtained by the regression and modifying the sequence accordingly. Through the merging and the subsequent data refinement (using the same line of reasoning applied to the TX), the i2s coverage too reaches a total coverage of 100% as shown in fig. 5.4.

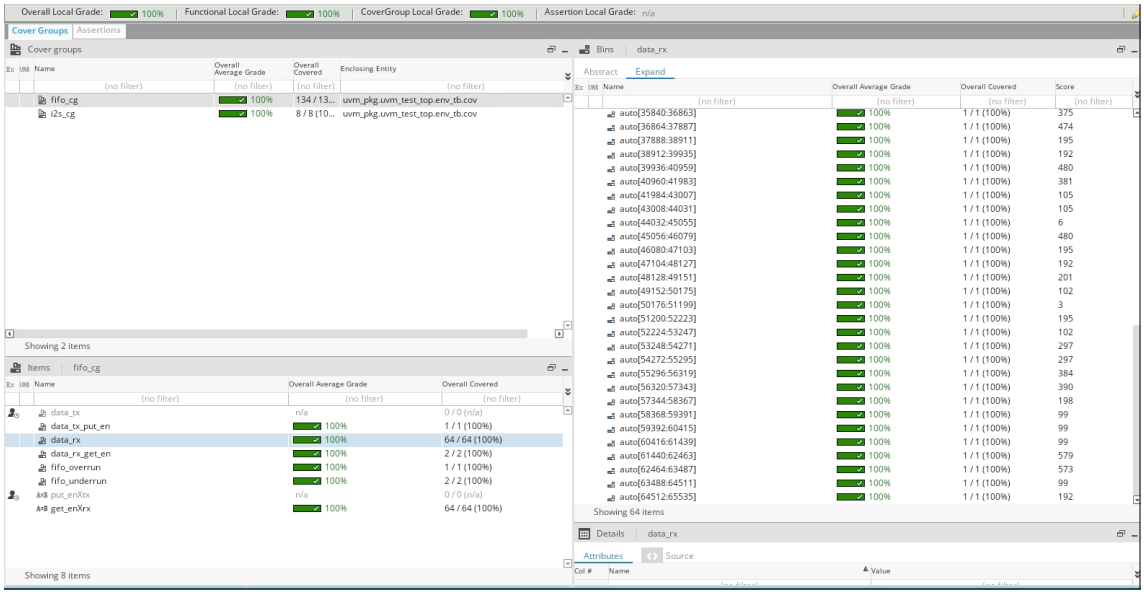

Figure 5.4: Rx coverage after merging and refinement

The merging of the rx and tx final reports shows the total obtained coverage of both the working modes TX and RX fig. 5.5.
In this report an almost perfect coverage can be observed, with a cover percentage equal to 100% for most of the metrics. This is ideally the goal that each verification process has, but unfortunately it is not always reachable in a reasonable amount of time.
For the sake of completeness, a final report with all the metrics is shown in fig. 5.6. As it can be seen, all the metrics except for the top entity have a coverage equal to 100%. Analyzing more in detail, it is shown that the protocol unit is the one with a not perfect coverage. The Protocol has the FSM tested completely, but 2 expressions not. Those two expressions represent two limit scenarios that do not happened during the regression.
Even with this small lack the verification process can be concluded, with the 99.86%

of the DUT tested correctly.

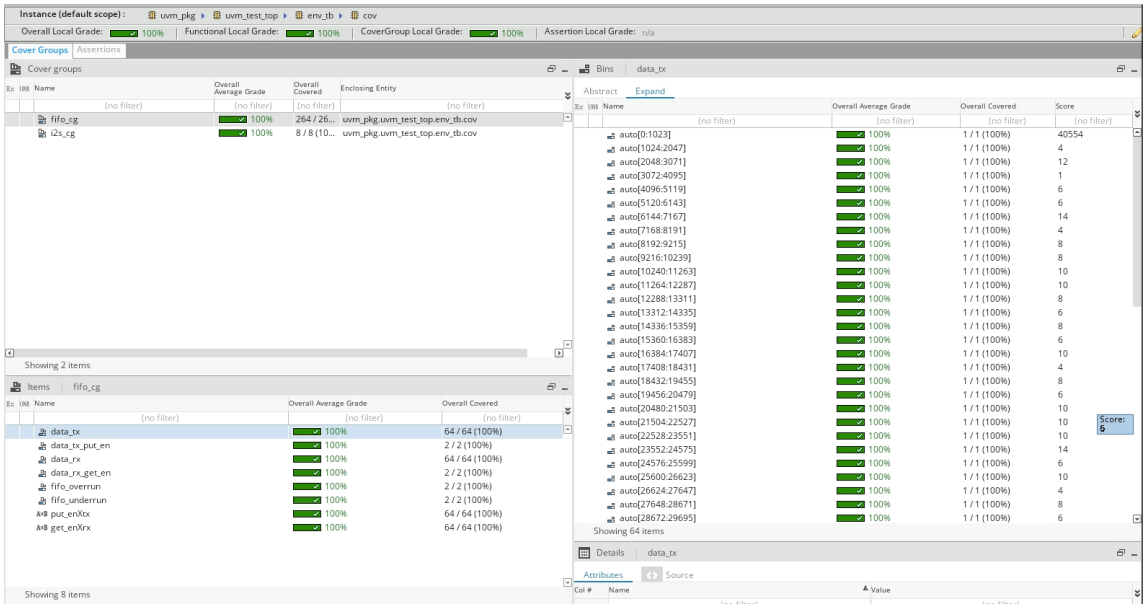

Figure 5.5: Final coverage report

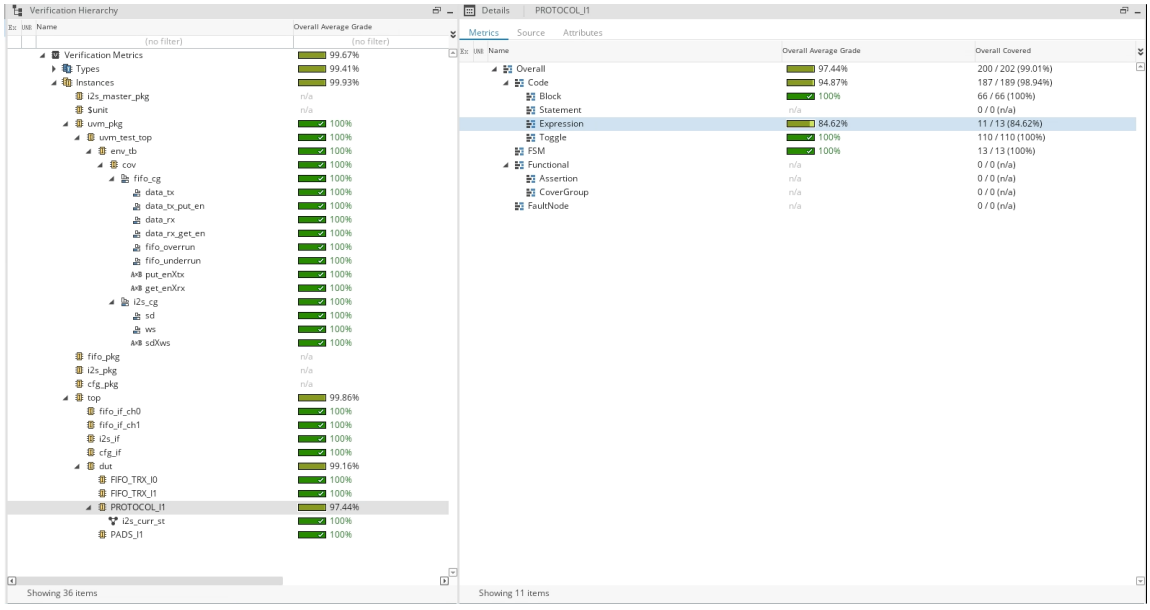

Figure 5.6: Final tx and rx merged report

Chapter 6

Conclusions

This thesis work proposed the development of a UVM environment for the verification of a I2S protocol.
Due to the continuous development of electronic systems and their increasing complexity, verification has gained exponentially more importance.
The verification of the protocol started from its analysis and in depth study as detailed in chapter 2, allowing a full understanding on its basic, most important principles and working scenarios. The first step has been to explicitly identify and declare all the protocol functionalities to be tested. Once the objective has been declared, the code development part started. By using the UVM library of SystemVerilog, an ad-hoc environment has been created. As described in chapter 3, the key principles of reusability and modularity have been followed in order to create a flexible and easy to be maintained architecture.
To evaluate all the decided functionality of the two working mode, namely TX and RX, seven different tests have been created, their description is provided in chapter 4 . chapter 5 presents an in depth discussion on the results obtained using cover groups, cover points and assertions to correctly understand which parts of the DUT has been tested. After the test of TX and RX functionalities on their own, the results obtained were merged and refined allowing the correct evaluation of the obtained data. At the end, the total coverage of the DUT has reached a percentage of 99.86% very near the ideal 100%.
After the verification of the I2S master block developed by TDK InvenSense, future works may focus on develop a test case able to certainly cover also the last 0.14% not covered. After obtaining this result, the verification of a complete I2S module could lead me to applying the experience gained in this project on an even more complex one.

Bibliography

[1] *e Reuse Methodology*. Jan. 2023. URL: https://en.wikipedia.org/wiki/E_Reuse_Methodology (cit. on p. 3).

[2] *VMM*. URL: https://semiengineering.com/knowledge_centers/eda-design/verification/methodology/vmm/#:~:text=Created%20by%20Synopsys%2C%20VMM%20harnesses%20language%20features%20such,SystemVerilog%2C%20provides%20industry%20best%20practices%20developed%20since%202005 (cit. on p. 3).

[3] Cadence Design Systems. *Open Verification Methodology*. URL: https://www.cadence.com/en_US/home/alliances/standards-and-languages/open-verification-methodology.html (cit. on p. 3).

[4] Cadence Design Systems. *Universal Verification Methodology*. URL: https://www.cadence.com/en_US/home/alliances/standards-and-languages/universal-verification-methodology.html (cit. on p. 3).

[5] Siemens EDA. *Universal Verification Methodology UVM Cookbook*. 2021 (cit. on pp. 3, 17).

[6] Cadence. *SystemVerilog Accelerated Verification with UVM (Engineer Explorer Series) v1.2.5*. Cadence_Learning_&_Support. Online course (cit. on p. 4).

[7] Ben Miller. *The I2S Protocol and Why Digital Audio is Everywhere*. Apr. 2022. URL: https://www.keysight.com/blogs/en/tech/bench/2022/04/29/the-i2s-protocol-and-why-digital-audio-is-everywhere (cit. on pp. 6, 7).

[8] *I2S bus specification*. White Paper. NXP, Feb. 2022 (cit. on pp. 7, 8).

[9] Alessio Pelle. *I2S_MASTER_TXRX Functional Specifications*. TDK Invensense Private Paper. TDK Invensense, Feb. 2024 (cit. on pp. 9, 11, 12).

[10] *UVM Tutorial*. URL: https://www.chipverify.com/uvm/uvm-tutorial (cit. on p. 13).

[11] *Base Classes*. URL: https://www.chipverify.com/uvm/base-classes (cit. on p. 14).

[12] *UVMEnvironment*. URL: https://www.chipverify.com/uvm/uvm-environment (cit. on p. 16).

[13] Cadence. *SystemVerilog Accelerated Verification Using UVM*. Apr. 2023 (cit. on pp. 18, 19).

[14] *UVM Scoreboard*. URL: https://vlsiverify.com/uvm/uvm-scoreboard/ (cit. on p. 19).

[15] Mentor Graphics Corporation. *Coverage Cookbook*. 2019 (cit. on pp. 58, 59).

[16] Cadence. *Xcelium Integrated Coverage v20.09(Online)*. Cadence_Learning_&_Support Online course (cit. on pp. 59, 60).

[17] *Assertion Based Verification*. URL: https://www.chipverify.com/verification/assertion-based-verification (cit. on p. 60).

[18] *Assertions in SystemVerilog*. URL: https://verificationguide.com/systemverilog/systemverilog-assertions/ (cit. on p. 61).