POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Development of System Verification Methods for Aircraft Electronic Systems

Supervisors

Prof. Mario Roberto CASU

Eng. Riccardo STICCA

Candidate

Lucia VENCATO

April 2024

Abstract

In aviation industry, it is essential to guarantee the safety and reliability of aircraft equipment since hardware failures can have disastrous effects. As a consequence, the verification process is indispensable, providing a methodical and rigorous approach to ensure that hardware components meet specified requirements and adhere to stringent safety guidelines. This thesis work was conducted in Leonardo Electronics in the context of hardware verification. The primary objective of this research is to study and explore the Universal Verification Methodology (UVM), with the goal of developing a comprehensive testbench for a Universal Asynchronous Receiver/Transmitter (UART) transmitter device. UVM relies on System Verilog language, which exploits Object-Oriented Programming constructs, and includes a set of pre-defined classes and methods that allow to create scalable, reusable, and maintainable verification environments. Initially, the primary UVM components were chosen, and the testbench infrastructure was developed using Siemens EDA's UVM Framework (UVMF) code generator. UVMF requires configuration files, written in yaml language, to create the testbench. Therefore, each component was correctly described following yaml guidelines. After obtaining the basic testbench, further modifications were implemented to customize the behavior of individual components in order to meet the specific requirements of the Device Under Test (DUT). The developed testbench includes different UVM components: an agent responsible for transmitting input data to the DUT, a second agent tasked with collecting output data from the DUT, a predictor utilized for calculating the golden output values, and a scoreboard utilized to verify the correspondence between the golden values and the actual outputs. Simulations were conducted to verify the correct behaviour of the developed testbench and to test the uart transmitter device under various configurations. QuestaSim and Visualizer, provided by Siemens EDA, were the software tools used to run simulations.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In aviation, where hardware failures can have catastrophic consequences, it is critical to ensure the safety and reliability of airborne equipment. Therefore, the verification process is essential, acting as a methodical and thorough means of verifying that hardware components fulfill requirements and conform to strict safety standards.
In hardware verification, Universal Verification Methodology (UVM) emerges as a powerful and standardized framework which offers a methodology for creating efficient and reusable verification environments, allowing engineers to streamline the verification process and increase productivity.
This thesis project was conducted in Leonardo Electronics Caselle Plant, which is involved into the design and production of Avionic Systems since 1950. In particular, their main focus is on Hardware Safety Critical products, and consequently RTCA DO-254 standard is a key point in the verification flow.

1.1 DO-254 Standard

DO-254, "Design Assurance Guidance for Airborne Electronic Hardware," is a standard developed by RTCA (Radio Technical Commission for Aeronautics) and it defines a comprehensive set of guidelines for the development and verification of electronic hardware in airborne systems [1].
Key aspects of DO-254 are:

- Life Cycle Approach: it covers the complete life cycle of airborne electronic hardware, from design to development, verification, and then production.
- Verification Process: it confirms that the hardware component developed meets the requirements. It includes conducting reviews throughout the design process.

- Tool Qualification: it specifies guidelines for ensuring that the tools used do not introduce errors or uncertainties.

DO-254 standard introduces five levels of classification, know as Design Assurance Levels (DAL), used to define the accuracy that should be applied to the design and verification processes. DALs are shown in Table 1.1.

Level	Failure Conditions	Effect of anomaly	Example
A	Catastrophic	Prevent continued safe flight and landing	Fly-by-wire
B	Hazardous/Severe	Serious or potentially fatal injuries to a small number of occupants	Fuel management
C	Major	Discomfort to occupants, possibly including injuries	Pilot/ATC communication
D	Minor	Some inconvenience to occupants	Flight data recorder
E	Not Relevant	No effect on aircraft operational capability or pilot workload	Entertainment system

Table 1.1: DO-254 Design Assurance Levels

Since in Leonardo Caselle Plant Hardware Safety Critical products are developed, they must be compliant with DAL A. DO-254 flow is shown in Figure 1.1.

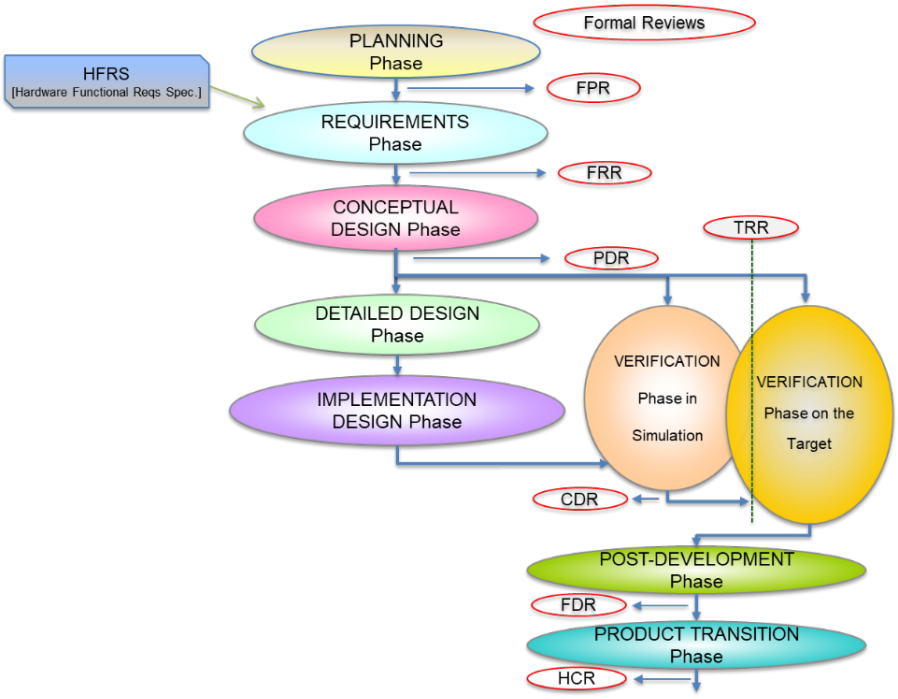

Figure 1.1: DO-254 Flow

1.2 Goal of the thesis

According to the DO-254 flow, this thesis project is situated within the phase denoted as "Verification - Phase in Simulation".
The objective is to research and construct a UVM (Universal Verification Methodology) verification environment for a UART transmitter device. Furthermore, the purpose of this research is to determine whether UVM methodology provides any advantages over the company's current VHDL-based testbench development.

1.2.1 Software tools

For the simulation of developed testbench two software were used, both provided by Siemens EDA:

- QuestaSim: it is used for simulation of hardware description languages, and it offers advanced debugging capabilities.
- Visualizer: it can be considered a QuestaSim upgrade, which also includes some UVM specific debug features.

Furthermore, material for studying SystemVerilog and UVM methodology was provided by the *Verification Academy* [2], an online platform by Siemes EDA which offers learning resources and a community ready to answer to any verification related question.

Chapter 2

Background

2.1 SystemVerilog

SystemVerilog is a unified hardware design and verification language. It is an extension of the Verilog hardware description language, and it was developed to address the increasing complexity of modern digital designs [3].
SystemVerilog introduces features such as:

- Extended data types: it adds some new data types such as *bit*, *byte*, *shortint*, *int*, *longint*, *real*, and others, providing more flexibility in data representation.
- Interfaces: used to encapsulate functionality between blocks.
- Object-Oriented Programming (OOP): it allows the use of classes, inheritance, and polymorphism, enabling the development of more modular, reusable, and maintainable code during both design and verification.
- Constrained Random Generation: it introduces the *rand* keyword and constraints can be applied to control the distribution of random values.
- Assertions: used to specify formal properties that must hold true during simulation.
- Functional coverage: a measure of how well the design has been verified with respect to specific features.

2.2 Universal Verification Methodology

Universal Verification Methodology (UVM) is a standard methodology used to verify digital designs and System-on-Chip (SoC). Developed by Accellera in 2011,

it offers a systematic and scalable approach to verification, promoting reusability, maintainability, and collaboration within the design and verification community [4].
UVM is derived from previous verification standards: OVM, VMM and eRM.
OVM (Open Verification Methodology) was introduced in 2008 and it is an open-source verification methodology based on SystemVerilog. VMM (Verification Methodology Manual) is the SystemVerilog based verification methodology created by Synopsis. eRM (e Reuse Methodology) is an aspect-oriented language that was developed by Verisity. Each of these earlier standards contributed to a universal best-way to simulate and verify a wide range of integrated circuit designs [5][6].
UVM is built on SystemVerilog's object-oriented programming constructs and includes a set of pre-defined classes and methods that allow users to create scalable, reusable, and maintainable testbench structures.
UVM verification topology is divided in two parts:

- Tests and Stimulus: they can be constantly created during the project
- Testbench: it changes slowly over the project

A generic view of a UVM verification environment can be seen in Figure 2.1.

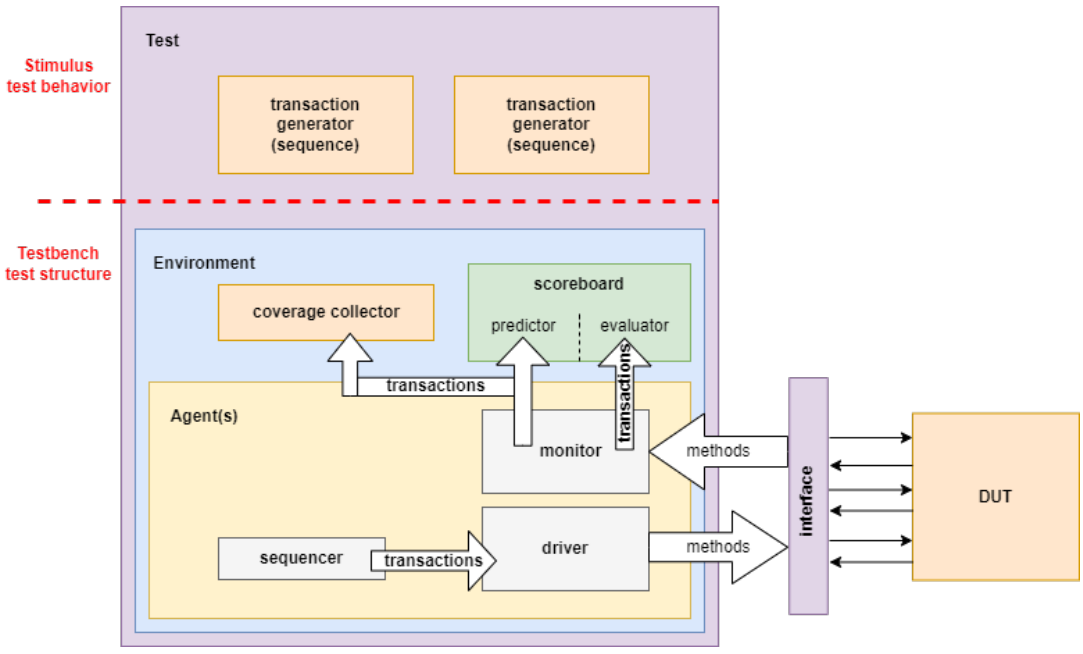

Figure 2.1: UVM Verification Environment

2.2.1 UVM Testbench

Building blocks of a UVM testbench are class-based. These are called UVM Components, and they are component objects extended from the uvm_component base class [5].

Transaction: also known as Sequence Item. It is a high-level data set with standard methods and collects information that is transferred between components.

Sequence: it is a constrained random stimulus generator that creates transactions.

Sequencer: it is responsible for routing transactions to a driver using built-in arbitration algorithms.

Driver: it receives transactions from the sequencer and drives them to the DUT (Device Under Test) through methods in a virtual interface. Most drivers have their Bus Functional Model (BFM) which converts transactions in signal activity.

Monitor: it is a passive component that samples DUT signals through interface methods. It is responsible of observing pins wiggle on a bus and converting them back into transactions.

Agent: it encapsulates and configures a sequencer, driver, monitor, coverage collector. It has two operating modes:

- Active: it provides stimulus to the DUT and monitor signal activity.
- Passive: it monitors signal activity, so it only observes transfers between RTL and broadcasts the observed transfers.

Scoreboard: it is an analysis component which collects transactions sent by a monitor and perform specific analysis computations to determine whether or not the design is functioning as expected. It is often split into two parts:

- **Predictor**: it receives the same stimulus transactions as the DUT and it implements a model of the DUT (also known as Golden Reference Model) to compute the expected values.
- **Evaluator**: the actual scoreboard, which performs the check on the expected values received by the predictor and the actual ones received by the monitor.

Coverage Collector: it is an analysis component that samples observed transactions and activity into SystemVerilog functional coverage groups. The coverage

data collected from each test is stored into a shared coverage database used to determine overall verification progress.

Environment: it encapsulates and configures one or more agents and analysis component.

An important feature of UVM is the **Factory** mechanism [5]. It improves flexibility since it allows a class object to be replaced with another one of a derived type, without changing code or structure of the testbench.

2.2.2 UVM Phases

UVM phases are needed to structure the simulation flow and provide synchronization for the components of the testbench. Each component follows a predefined set of phases and cannot proceed to the next phase until all components have completed their execution in the current one. The introduction of the object-oriented programming feature in SystemVerilog requires phases: Classes can be reused and created whenever it is required and, without a synchronization mechanism, it is possible that a component is called even if it has not been initialized yet, resulting in incorrect testbench outputs [7][5].
There are three groups of phases, which are reported in Table 2.1:

Phase Category	**UVM phase**
Build	build connect end_of_elaboration
Run	start_of_simulation run
Cleanup	extract check report final

Table 2.1: UVM Phases description

- **Build phases**: executed at the start of the simulation. Their objective is to construct, configure and connect the testbench component hierarchy. All build phases methods are functions and therefore they execute in zero simulation time. The *build* phase works top to bottom, so each component of a layer is constructed by the level above.

- **Run phases**: follow the build phases. During this phases time is consumed in running the testcase on the testbench
- **Clean-up phases**: results of the testcase are collected and reported. These phases extract information from scoreboards and functional coverage monitors and are implemented as functions (so zero time to execute). They work from the bottom to the top of the component hierarchy.

2.2.3 Transaction Level Modeling

Transaction-Level Modeling (TLM) is a methodology that allows different components of the testbench to communicate and exchange information at a higher level of abstraction than signal-level interactions. In TLM data are represented as transactions, which flow in and out components through ports called TLM interfaces:[7]

- **port**: object that define the set of methods available (API) to do the communication (such as calling *get()*).
- **export**: object that contains the implementation of the port's method (such as *get()* body).
- **analysis port**: used for broadcasting transactions

"port" must be connected to exactly one "export" (Figure 2.2), while "analysis port" can be connected to any number of "analysis export" (Figure 2.3) [6].
TLM increases reusability and flexibility, because it allows to replace a component with another as long as it has the same TLM interface.

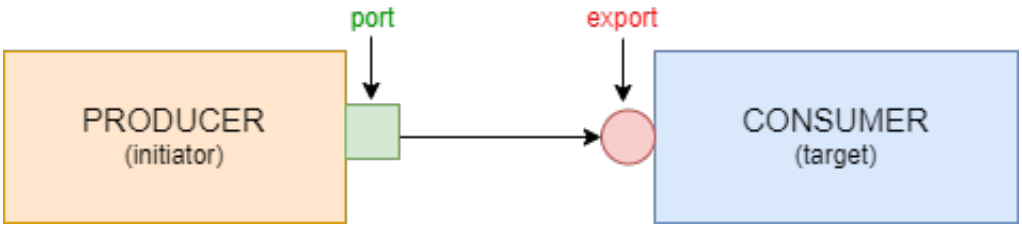

Figure 2.2: TLM interfaces

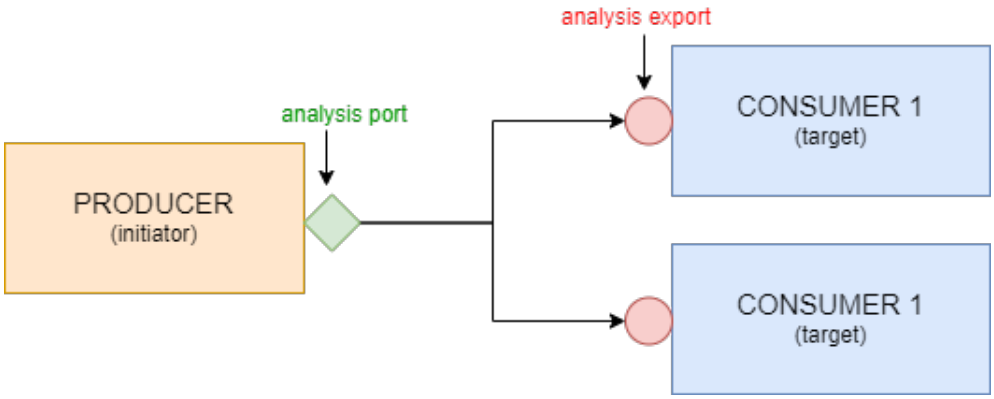

Figure 2.3: TLM interfaces

2.2.4 Testbench configuration

A testbench needs to be configured: there are some values that must be shared between different components and that can be set at run time. These values can be represented as SystemVerilog variables and can be organized into objects. Specifically, UVM includes a *Configuration Object* which is an efficient, reusable means for organizing configuration variables.

2.2.5 UVM Messaging

UVM provides an infrastructure for printing messages in a testbench, which is included in all UVM components. There are four message types supported, reported in Table 2.2 [5]:

Type	**Usage**
UVM_INFO	Informative messages
UVM_WARNING	Warning message
UVM_ERROR	Error message
UVM_FATAL	Error message that results in an exit from simulation

Table 2.2: UVM Messages

UVM messages can have associated actions, allowing control over where they are sent, and whether they stop or finish the simulation. As a result, they are an effective tool for managing testbench simulations.

2.3 UVM Framework

Siemens EDA offers an open-source package, UVM Framework (UVMF), that provides a reusable UVM methodology and a code generator for easier testbench creation [8]. It provides benefits such as [9]:

- **Schedule reduction**: for beginners learning UVM can be challenging because of its extensive features and concepts, in fact a lot must be learned before a production environment is operational. UVMF provides a starting point to learn UVM and create UVM verification environment. In fact, it automates the development of the infrastructure and interconnections for interface and environment package and project benches. Thanks to this, verification engineers can immediately work on implement design-specific functionalities.

- **Reuse methodology**: it supports component level verification reuse across projects and environment reuse from block through chip to system level simulation.

- **Consistency**: it decreases integration effort when reusing verification components.

In order to use UVMF, YAML (see subsection 2.3.1) configuration files for interface packages, environment packages and project benches have to be created.
After the main component of the verification environment have been characterized, they are sent as input to a python script that generates the relative UVM testbench. Once the infrastructure is generated, it is possible to manually change the code, in order to add the connection to the DUT and all the needed design-specific functionality.
UVMF split the environment in two parts:

- **HDL**: the HDL (Hardware Design Language) side is written as synthesizable modules or interfaces, it is timed (synchronizes with DUT) and drives (or monitors) the pins of the DUT.

- **HVL**: the HVL (Hardware Verification Language) side is written in object-oriented UVM code, it is untimed (does not have delays), it passes transactions into interface methods and it does not directly assign to DUT's ports.

Due to this separation, driver and monitor inside an agent are respectively divided in two components:

- Monitor proxy and Driver proxy: UVM components, they are class-based objects and perform transaction level operations.

- Monitor BFM and Driver BFM: SV interfaces or modules, they communicate with the DUT and perform signal to transaction conversion (Monitor BFM) and transaction to signal conversion (Driver BFM).

A generic agent architecture can be seen in Figure 2.4.

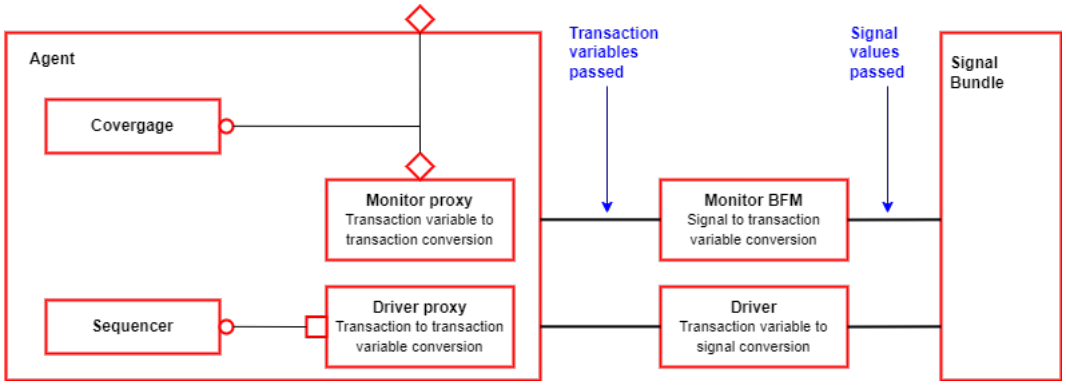

Figure 2.4: UVMF agent architecture

A generic UVMF environment architecture can be seen in Figure 2.5.

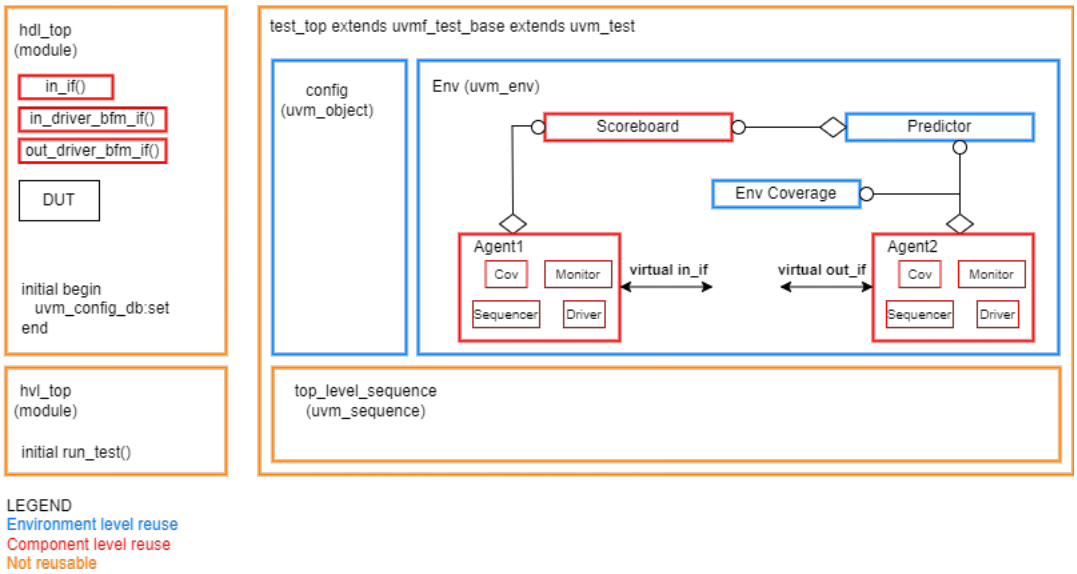

Figure 2.5: UVMF environment architecture

2.3.1 YAML

YAML is a human friendly data serialization standard used for configuration files and the name stands for "YAML Ain't Markup Language". It is used in the generation of a UVMF verification environment due to its ease of translation of data structures describing UVMF hierarchy and attributes [10].
All UVMF YAML has to be presented as part of a specific top-level format, shown in the following code taken from *UVMF YAML Reference Manual* [10].

```
uvmf:
    interfaces:
        "<interface_nameA>"
        <properties>
        "<interface_nameB"
    util_components:
        "<util_componentA>"
        <properties>
    environments:
        "<env_nameA>"
        <properties>
        "<env_nameB>"
        <properties>
    benches:
        "<bench_nameA>"
        <properties>
        "<bench_nameB>"
        <properties>
    gloabal:
        <properties>
```

Each named subsection represents a YAML data structure, with the following content:

- `interfaces`: information about an interface's name, transaction data, ports and configuration. It creates classes (agent, driver, monitor, transaction, interface level sequences), package, BFMs and compilation files.
- `util_component`: it is used to repesent UVMF predictor, scoreboard and coverage components.
- `environments`: information about an environment's name, instantiated components, TLM connections and configuration. It creates classes (environment, predictors, environment level sequences, coverage components), package and compilation files.

- `benches`: information about a bench's name, top-level environment, passive/active settings for BFMs and data for driving reset and clock. It creates classes (top level test and top level virtual sequence) package, modules (hdl, hvl) and compilation files.
- `global`: it is used to provide information that can be applied to all other objects.

2.4 UART

Since the DUT analyzed in the developed testbench is a UART device, fundamentals of this protocol are discussed.
UART, Universal Asynchronous Receiver Transmitter, is a serial communication protocol used in embedded systems, microcontrollers, and other electronic devices for transmitting data between devices. It allows two devices to communicate with each other serially over two wires, one for transmitting data (TX) and one for receiving data (RX). It operates asynchronously, so there is no clock signal shared [11].
UART uses character-based transmission, so each transfer consists of a frame. A frame has a defined format, as shown in Figure 2.6:

- START BIT (1 bit) : it is always a logic 0. It is used to indicate the start of a transmission.
- Character (n bits) : bits of a character to be sent. Common values are: 6, 7, 8, 9 bits. Character bits are transmitted LSB (least significant bit) first.
- PARITY BIT (0 or 1 bit) : it is present if parity is enabled for the transmission. If parity is present, it could be even or odd. If it is EVEN, the total number of bits equal to 1 (including the parity bit) should be even, while if it is ODD the total number of bits set at 1 should be odd.
- STOP BIT (1 or 2 bits) : it is always a logic 1. It is used to indicate the end of a transmission and it can be configure as 1 bit or 2 bits.

Figure 2.6: UART data frame [11]

Clock synchronization can be done at the beginning of a frame since the START BIT of a new frame always triggers a 1 to 0 transition, either during an idle time

or after the previous phase.
Transmitter and receiver should agree on different parameters: baud rate (bits per second (bps)), that specifies the speed at which data is transmitted and received, number of bits in a character, presence and kind of parity and number of stop bits.

Chapter 3

UVM Verification Environment

The objective of the presented thesis work is to develop a UVM verification environment for testing a UART transmitter device. UVM Framework was used to build the initial infrastructure, since it helps instantiating classes and connections between them.

3.1 Device Under Test

The Device Under Test (DUT) for the developed testbench is a UART transmitter device. Its diagram is shown in Figure 3.1.

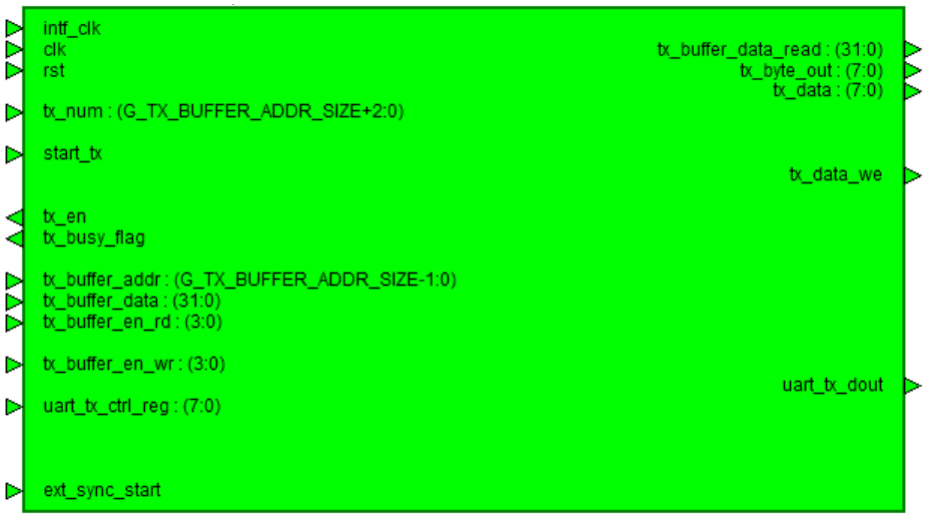

Figure 3.1: UART transmitter

UART tx receives parallel data (4 bytes word) on the input port `tx_buffer_data` and serializes them on the output port `uart_tx_dout`. Once data are received on the input port, they are stored in a memory at a specified address, waiting to be sent. UART transmitter interface provides a set of registers, the ones involved in the developed testbench are the following:

- `tx_num`: 8-bit register used to start a UART transmission. It stores the number of bytes to be sent. If it is not equal to zero, then it starts a transmission.
- `tx_busy_flag`: 1-bit flag, it remains at 1 when a transmission is in progress.
- `start_tx`: 1-bit register, it goes high for one clock cycle to indicate that a transmission has started.
- `tx_buffer_addr`: 8-bit register, it represents the memory address where data are written.
- `tx_buffer_data`: 32-bit register, it stores input data to be sent over the uart.
- `tx_buffer_en_wr`: 4-bit register, it is used to mask bytes that are not sent over the uart.
- `uart_tx_ctrl_reg`: 8-bit register, it contains uart configuration. It is divided in sub-fields that allows to: enable a parity, select the parity (odd or even), set the number of stop bits and select a baud rate.
- `uart_tx_dout`: 1-bit register, it outputs the data received serializing it.
- `tx_en`: 1-bit register, it is equal to 1 when UART is transmitting.

Once the right behavior of the DUT was carefully analyzed, it was possible to begin developing the testbench.

3.2 Development Flow

The flow for developing a UVM testbench, using the UVM Framework, can be divided into four steps:

1. Design the verification blocks to be generated
2. Translate the design into yaml
3. Generate the verification blocks using UVMF and simulate the generated code
4. Add custom code to the generated one

These steps are discussed in the following subsections.

3.2.1 Verification blocks design

The starting point is designing the verification blocks required for the testbench. First, it was necessary to identify how many interfaces there were for the UART tx, from which the number of agents is decided. Consequently two agents were needed in the environment. One is used for the UART input interface (`UART IN_AGENT`) and the other for the UART output interface (`UART OUT_AGENT`). A predictor and a scoreboard were also instantiated as main components: `UART PREDICTOR` is used as Golden Model to produce expected values, while `UART SCOREBOARD` receives actual values from (`UART OUT_AGENT`) and compares them.
Figure 3.2 illustrates the general block diagram for the UART transmitter device.

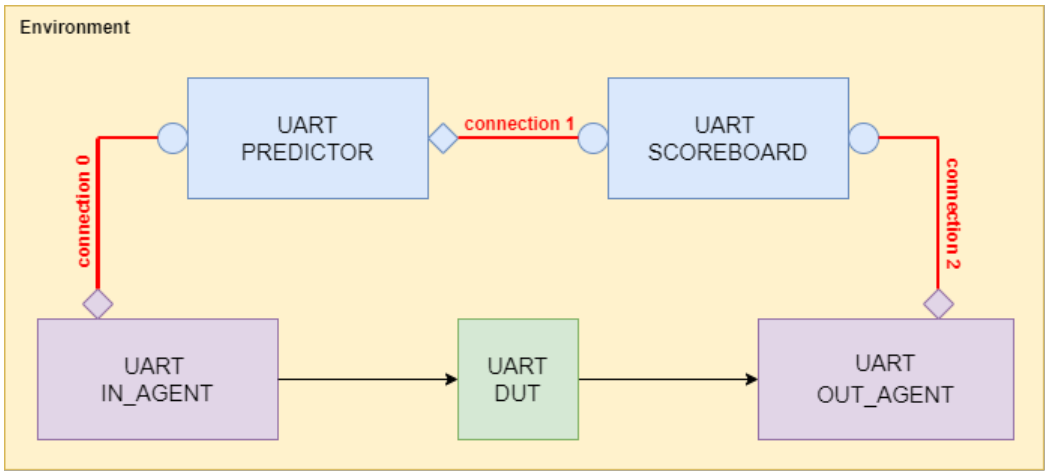

Figure 3.2: UART Environment Block Diagram

After the design was completed, the blocks must be accurately translated into the YAML language in order, for the generator, to properly build the environment.

3.2.2 Design to YAML translation

Each of the designed components were translated in YAML language following guidelines in the reference manual [10]. The following files were created:

- **UART_bench.yaml**: configuration file used to generate the UVMF top level testbench. It declares the name of the top level environment to instantiate and specifies agents (declaring them as ACTIVE or PASSIVE). It also defines period and phase offset of the clock and the duration of the reset.
- **UART_environment.yaml**: configuration file used to generate the environment level classes. It defines agents, predictor and scoreboard that need to be instantiated. It declares TLM connections between components (connection 0,

connection 1, connection 2 in Figure 3.2). It also specifies environment level configuration variables and parameters for the environment class.

- **UART_in_interface.yaml**: configuration files that captures pin information and transaction information for the interface `UART_IN_AGENT`. It defines interface ports (signal names, directions and widths), configuration variables and parameters for the agent. It also includes variables to be used by the transaction class. For `UART_IN_AGENT` the transaction variables are: `tx_buffer_addr, tx_buffer_data, tx_buffer_en_rd, tx_buffer_en_wr, tx_num`.

- **UART_out_interface.yaml**: it is the same as UART_in_interface.yaml, but contains information for the `UART_OUT_AGENT`. For this class there is just one transaction variable: `uart_tx_dout`.

- **UART_util_comp_predictor.yaml**: configuration file for describing the predictor to be used in the environment. It defines analysis port and analysis exports for the component.

UVMF generator used these files to build the infrastructure, from which further custom functions were added.
Since all the components in the testbench need to know the configuration of the UART transmitter device, configuration variables were declared in the files describing the UART_environment, UART_in_interface and UART_out_interface. They are listed in Table 3.1:

Configuration Variables	
`uart_config`	7-bit, it contains UART configuration
`stopBit`	2-bit, it ccontains the number of stop bit for the data frame
`parity_enabled`	1-bit, it is equal to 1 if parity is enabled
`parityBit`	1-bit, it represents the parity (odd or even)
`baudRate`	integer, it contains the baud rate in bit per seconds (bps)
`bit_transmission_cc`	integer, it is the number of clock cycle needed for transmitting a single bit
`startBit`	1-bit, it is always equal to 0
`data_bits`	8-bits, it contains the number of bits to be transmitted
`data_frame_length`	12-bit, it represents the maximum size of a data frame

Table 3.1: UART configuration variables

3.2.3 Verification blocks generation and code simulation

In order to generate testbench, environment and interface code a python file was executed. The file used is *yaml2uvmf.py*, which is provided by UVM Framework. This files receives in inputs all the yaml configuration files needed. In order to automate the process a batch file (*run_yaml_uvmf_scripts.bat*) was written: it

sets environment variable for the installation path of UVM Framework and calls python on the configuration files.
If code generation returned with zero errors a new directory is created. It is named `uvmf_template_output` and it is divided into two sub-directories:

- `project_benches`: it contains top level UVMF testbench files plus scripts for compiling and running simulations.
- `verification_ip`: it contains two directories: `environment_packages` and `interface_packages`. Each of them includes files and directories for respectively environment and interfaces classes.

Before making any manual code modifications to the generated one, a first simulation was run. This allows for potential issues to be fixed by making changes to the yaml files and then regenerating the code.

3.2.4 Custom code addition

Once the code was successfully generated, DUT functionalities were added manually. The UVM Framework adds some labeled comments (`UVMF_CHANGE_ME`) to the generated code to help identify areas that require additional coding.
The initial modification consisted in instantiating the DUT and connecting it up to the agent BFMs, clock and reset. *hdl_top.sv* is the file to be changed, since it contains: Monitor and Driver BFMs for the UART, clock and reset generators and the DUT. UART instance was added and ports were wired up to the corresponding agent interface. In the following, code for connecting DUT to the correct components is reported. It brings attention to the way the signals were organized, since it is noticeable which signal was connected to the UART IN AGENT and which one to the UART OUT AGENT interface.

Listing 3.1: DUT connection

```
1  uart_tx_top DUT(
2    .clk                (UART_in_agent_bus.clk),
3    .rst                (UART_in_agent_bus.rst),
4    .ext_sync_start     (UART_in_agent_bus.ext_sync_start),
5    .start_tx           (UART_in_agent_bus.start_tx),
6    .tx_buffer_addr     (UART_in_agent_bus.tx_buffer_addr),
7    .tx_buffer_data     (UART_in_agent_bus.tx_buffer_data),
8    .tx_buffer_en_rd    (UART_in_agent_bus.tx_buffer_en_rd),
9    .tx_buffer_en_wr    (UART_in_agent_bus.tx_buffer_en_wr),
10   .tx_num             (UART_in_agent_bus.tx_num),
11   .tx_busy_flag       (UART_out_agent_bus.tx_busy_flag),
12   .uart_tx_ctrl_reg   (UART_in_agent_bus.uart_tx_ctrl_reg),
13   .tx_byte_out        (UART_out_agent_bus.tx_byte_out),
14   .tx_data            (UART_out_agent_bus.tx_data),
```

```
15      .tx_data_we             (UART_out_agent_bus.tx_data_we),
16      .tx_en                  (UART_in_agent_bus.tx_en),
17      .tx_buffer_data_read    (UART_out_agent_bus.tx_buffer_data_read),
18      .uart_tx_dout           (UART_out_agent_bus.uart_tx_dout),
19      .intf_clk               (UART_in_agent_bus.clk)
20    );
```

Subsequent changes included: addition of protocol specific information to driver and monitor BFMs for both agents, addition of DUT specific behavior to the predictor and modification to the scoreboard in order to obtain custom report messages. Furthermore, sequences and test were modified and extended in order to exercise DUT functionality.
Modifications to each UVM component and file are described in details in the next sections.

3.3 UART Top Level Testbench

Top level testbench is responsible for instantiating the UART environment, allowing for top-down configuration of the UART environment, which then configures UART agents. It provides a default sequence along with a default test to run. The code is generated from the bench configuration file *UART_bench.yaml*, and it is specific to the DUT being tested, thus it is not reusable.
UVM components that characterize test behavior are detailed in the following subsections. These components include the type of transactions used, the test run, and the sequences used to generate desired stimulus.

3.3.1 UART Test Top

UVMF generates base tests for simulations, therefore the file *test_top.svh* was edited to customize the test and specify the setup for the UART environment.
This file instantiate the class `test_top` (extending `uvmf_test_base`) which contains the top level configuration and top level environment for the project. Function `build_phase` of this class is used to construct the configuration and environment classes. After their creation, the following operation are performed: Monitor and driver BFMs virtual interface handles are passed into agents and then active or passive state for each agent are set.
This function was modified specifically to configure the UART Environment and UART Agents' configuration variables correctly. Among the configuration variables listed in Table 3.1, the one which can be configured by the user for running the test is: `uart_config`. UART transmitter configuration is stored in this vector variable, where each bit denotes a parameter:

- `uart_config[7]`: UART transmitter enable.
- `uart_config[6]`: UART transmitter parity selector.
- `uart_config[5]`: UART transmitter parity enable.
- `uart_config[4]`: UART transmitter stop bit selector.
- `uart_config[3:0]`: UART transmitter baud rate selector.

The desired value for this variable is read from a text file: the user can edit the custom file *uart_configuration.txt*, in order to change the value of the `uart_config`. All other defined configuration variables are set correctly based on this value.
The UART Environment `build_phase` begins when the one of this class concludes, enabling the building of UART Agents.

3.3.2 UART Transactions

A transaction is used for gathering data that are communicated between UVM components. The developed testbench includes two type of transactions, one for each UART interface: `UART_in_transaction`, which collects signal of the UART input interface (`UART_in_agent`), and `UART_out_transaction`, that collects signal of the UART output interface (`UART_out_agent`).
Transaction variables of the two classes are shown in Table 3.2 and Table 3.3.

UART In Transaction
tx_buffer_addr
tx_buffer_data
tx_buffer_en_rd
tx_buffer_en_wr
tx_num

Table 3.2: UART In Transaction

UART Out Transaction
uart_tx_dout

Table 3.3: UART Out Transaction

In `UART_in_transaction`, the only variable that can be randomized (since it is declared as rand) is `tx_buffer_data`, in order to generate random data input for the DUT.
For these components, UVMF generates the files *UART_in_transaction.svh* and *UART_out_transaction.svh*. In both of them, the respective class is instantiated and it is extended from the `uvmf_transaction_base`. The class contains the corresponding variables and several methods for printing, comparing, copying, displaying variables in the waveform viewer of QuestaSim.
A transaction of type `UART_in_transaction` is utilized as a sequence item in the

interface sequences and is also provided to the UART Predictor to compute golden values.
On the other hand, a transaction of type `UART_out_transaction` is used for collecting actual data from the DUT and sending them the UART Scoreboard for comparison and verification.

3.3.3 UART Sequences

UART Sequences are used to generate stimuli to test the correct behavior of the UART transmitter device. UVMF generates base sequences for agents, environment, and bench, that can be modified as wanted to implement desired stimuli.
For this project, sequences of UART In Agent and the top level sequence used by `test_top` were modified. As a result, the test sequences created are of two types:

- Interface sequences: `UART_in_fill_sequence`, `UART_start_tx_sequence`
- Bench sequence: `UART_bench_sequence_base`

Interface sequences transmit a sequence_item to the driver, which requires four steps:

1. **Creation**: transaction of the correct type has to be constructed.
2. **Ready**: function `start_item()` is called with the transaction as argument. It blocks until the sequencer gives the sequence and sequence_item access to the driver.
3. **Set**: transaction is ready to be used. Its variable can be randomized (if specified as random) or set explicitly.
4. **Go**: function `finish_item()` is called with the transaction as argument. It blocks until the driver completes its side of the transfer protocol for the sequence_item.

Bench sequence is used by the top level test to run sequences, which consists of two steps:

1. **Sequence creation**: sequence of the proper type is constructed.
2. **Sequence starting**: sequence's `start()` method is called, having as argument the pointer to the sequencer that send the sequence_item to the driver. It calls the `body()` task of the sequence. Once this task completes, `start()` method returns.

UART In Fill Sequence

It is an interface sequence generated in the UART In Agent. It loads data into a memory for sending them to the DUT, as a consequence the transaction variable *tx_num* is set to logic 0.
File *UART_in_fill_sequence.svh* defines `UART_in_fill_sequence`, which extends the `UART_in_sequence_base` generated by UVMF. It contains a task `body()`, which is executed when the sequencer starts this sequence, and main code is reported below.

Listing 3.2: UART In Fill Sequence body() task

```
task body();
    req=UART_in_transaction#()::type_id::create("req");
    start_item(req);
    req.randomize();
    req.tx_num = 12'h000;
    req.tx_buffer_addr = 8'b00000000;
    req.tx_buffer_en_wr = 4'b1111;
    finish_item(req);
end task
```

This task constructs a transaction of the proper type. The transaction is then randomized; however, only the variables defined as *random* can be randomized; a constant value is assigned to the other ones. So, for a `UART_in_transaction`, only `tx_buffer_data` can be randomized. All transaction variables are defined and provided to the `UART_in_driver_bfm` using the `sequencer` and `UART_in_driver`.

UART In Start Tx Sequence

In this sequence, transaction variable `tx_num` has a value different from 0, thus it begins transmitting the loaded input data through the UART tx.
File *UART_in_start_tx_sequence.svh* defines `UART_in_start_tx_sequence`, and, as already stated for UART_in_fill_sequence, it contains a task `body()`. Main code is reported below.

Listing 3.3: UART In Start Tx Sequence body() task

```
task body();
    req=UART_in_transaction#()::type_id::create("req");
    start_item(req);
    req.randomize();
    req.tx_num = 12'h00C;
    req.tx_buffer_addr = 8'b00000010;
    req.tx_buffer_en_wr = 4'b1111;
    finish_item(req);
end task
```

Considering that it is an interface sequence created in the UART In Agent, as the UART_in_fill_sequence, the code is the same except for the variable values. In this sequence `tx_num` is equal to the hexadecimal value C, thus 12 bytes are sent to the DUT, *tx_buffer_addr* is set to decimal value 2, so the input data is written in a different memory address, and *tx_buffer_data* is still a random value. Thus, input data are written to memory and then a transmission begins.

UART Bench Sequence

It is a bench sequence that is used by `test_top` to start interface sequences described above.
File *UART_bench_sequence_base.svh* defines the `UART_bench_sequence_base` class, which extends `uvmf_sequence_base`. It instantiates sequences that are run, and it includes a `body()` task, whose code is reported in the following:

Listing 3.4: UART Bench Sequence body() task

```
1 virtual task body();
2    UART_in_agent_fill_seq = UART_in_agent_fill_seq_t::type_id::create
     ("UART_in_agent_fill_seq");
3    UART_in_agent_start_tx_seq = UART_in_agent_start_tx_seq_t::type_id
     ::create("UART_in_agent_start_tx_seq");
4
5    UART_in_agent_fill_seq.start(UART_in_agent_sequencer);
6    UART_in_agent_start_tx_seq.start(UART_in_agent_sequencer);
7 endtask
```

It first constructs desired sequences, in this case UART_in_fill_sequence and UART_in_start_tx_sequence. Then it starts both of them on the right sequencer, calling function `start()`, passing as argument the `UART_in_agent_sequencer`.
The two sequences are sent to the sequencer serially: `UART_in_agent_fill_seq` starts, and only when `start()` call returns, `UART_in_agent_start_tx_seq` can be transmitted.

3.4 UART Environment

UART Environment is the component responsible for configuring and connecting UART Agents, UART predictor and UART Scoreboard.
UVMF generated the file *UART_environment.svh*, which instantiate the class for this component (extending it from the `uvmf_env_base`). Functions `build_phase` and `connect_phase` are used to respectively build and connect components in the UART Environment. Connections are established via the analysis ports and exports of the various UVM components.

A more detailed schematic of the developed UVM environment could be seen in Figure 3.3, which was generated in Visulizer. In fact this software tool provides some UVM features, one of which is the possibility to examine a testbench schematic of the test that was run.

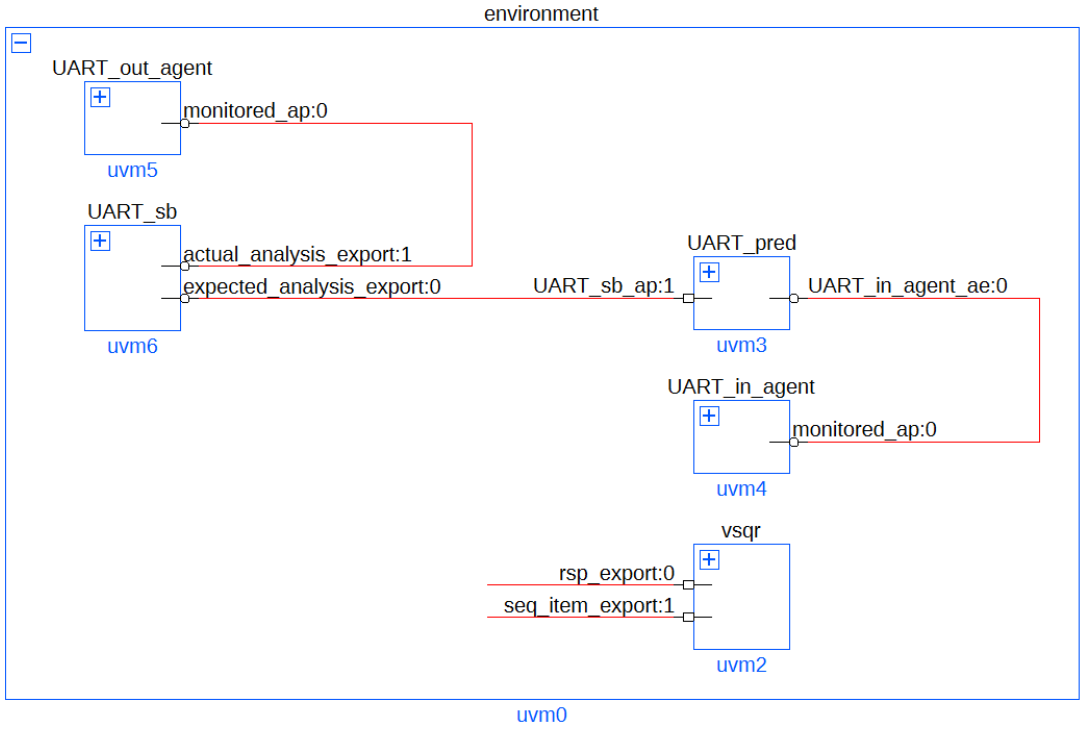

Figure 3.3: UART environment schematic from Visualizer

From Figure 3.3 it is possible to check if all the components, and connections between them, were instantiated correctly. Blocks in the picture represent the components that were changed to allow for testbench customization. Each instantiated component extends the corresponding UVMF base library.

3.4.1 UART In Agent

UART In Agent is the component that sends input data to both the DUT and the UART Predictor. It performs input protocol operations.
It is an ACTIVE agent, so it sends stimulus to the DUT as well as monitors signal activity. As a consequence, it is composed of both a Driver and a Monitor. Its internal components are shwons in Figure 3.4.

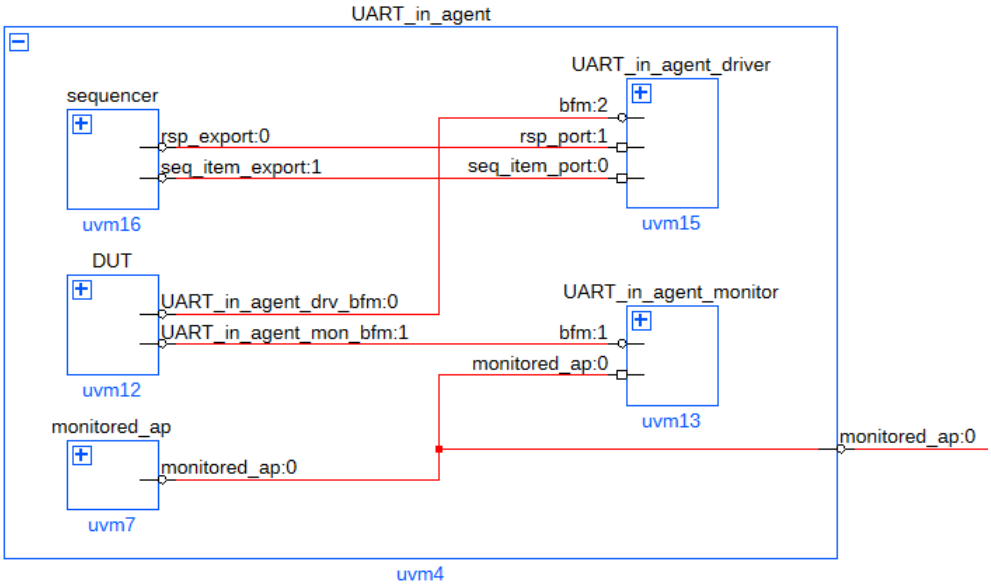

Figure 3.4: UART Agent In from Visualizer

Since it is an active agent it is composed of the following components, that were created and connected by the generator:

- Sequencer: no adjustments were made to this component because it is only used to send test sequences to the driver. It transmits sequences, which in turn constructs transactions of type `UART_in_transactions`, to the `UART_in_agent_driver`.
- Driver: driver BFM was changed in order to correctly perform protocol activities.
- Monitor: monitor BFM was updated to accurately capture data provided to the DUT.

This agent has a built-in analysis port named `monitored_ap`, which is the one responsible for sending transaction of type `UART_in_transaction()` to the predictor. As a result, "golden" output values can be calculated using the same transactions given as input to the DUT.

UART In Agent Driver

Due to the dual top architecture, which divides the testbench into an HDL and an HVL side, the Driver consists of two components: a driver class and a driver BFM interface. UVMF generator creates two different files: *UART_in_driver.svh* and

UART_in_driver_bfm.sv.

UART_in_driver.svh: it defines the class `UART_in_driver`, which is the one that passes transactions from the sequencer to the BFM driver interface. This class contains functions to set an handle to the Driver BFM and configure it, and a task that allows transactions to be passed to the Driver BFM.

UART_in_driver_bfm.sv: it defines the sv interface `UART_in_driver_bfm`, that performs the UART_in signal driving. Modifications were made to properly drive signals through the bus port connection.

Listing 3.5: modified task in UART in driver bfm

```
interface UART_in_driver_bfm #();
 task initiate_and_get_response(
       input UART_in_initiator_s UART_in_initiator_struct,
       output UART_in_responder_s UART_in_responder_struct
       );

    ext_sync_start_o <= ext_sync_start_i;
    uart_tx_ctrl_reg_o <= uart_config;
    while(tx_en_i == 1'b1) @(posedge clk_i);
    start_tx_o <= 1'b1;
    tx_buffer_addr_o <= UART_in_initiator_struct.tx_buffer_addr;
    tx_buffer_en_rd_o <= UART_in_initiator_struct.tx_buffer_en_rd;
    tx_buffer_en_wr_o <= UART_in_initiator_struct.tx_buffer_en_wr;
    tx_num_o <= UART_in_initiator_struct.tx_num;
    tx_buffer_data_o <= UART_in_initiator_struct.tx_buffer_data;

    @(posedge clk_i);
    start_tx_o <= 1'b0;
    repeat (2) @(posedge clk_i);

    responder_struct = UART_in_responder_struct;
  endtask
endinterface
```

The sv task reported above, `initiate_and_get_response()`, is used to initiate a transfer. It drives transactions received by the driver to the DUT, connecting each variable to a DUT port.
Timing for correctly sends data to the DUT were obtained by analyzing a UART simulation. When `tx_en` is equal to logic 0, it is possible to raise `start_tx` at logic 1 and set pin connection. At the following rising edge of the clock `start_tx` goes to 0, and then two clock cycles are waited, so that the monitor can sample correct data on the UART Agent In bus.

UART In Agent Monitor

As the driver, also the monitor is split in two components: monitor class and monitor BFM interface. These files are produced: *UART_in_monitor.svh* and *UART_in_monitor_bfm.sv*.

UART_in_monitor.svh: it defines the class `UART_in_monitor`, which broadcasts UART_in transactions via the agent's analysis port after receiving by the UART_in monitor BFM. It also records transactions for observing them in the waveform viewer of the simulator. It defines functions and task to configure and set the handle to the monitor BFM, and to be accessed by the interface.

UART_in_monitor_bfm.sv: it defines the sv interface `UART_in_monitor_bfm`, that performs the UART_in signal monitoring. Task `do_monitor` was modified to correctly sample signals on the bus, and the code is reported in the following:

Listing 3.6: modified task in UART in monitor bfm

```
interface UART_in_monitor_bfm #()( UART_in_if  bus );
  task do_monitor(output UART_in_monitor_s UART_in_monitor_struct);

    while(start_tx_i == 1'b0) @(posedge clk_i);
    UART_in_monitor_struct.tx_num = tx_num_i;
    UART_in_monitor_struct.tx_buffer_addr = tx_buffer_addr_i;
    UART_in_monitor_struct.tx_buffer_data = tx_buffer_data_i;
    UART_in_monitor_struct.tx_buffer_en_rd = tx_buffer_en_rd_i;
    UART_in_monitor_struct.tx_buffer_en_wr = tx_buffer_en_wr_i;

  endtask
endinterface
```

The signal monitoring implemented consisted in waiting for `start_tx` to go to logic 1. Once the transition happens, it is possible to read the signal on the bus in order to sample the correct transaction values.

3.4.2 UART Out Agent

UART Out Agent is the component that collects DUT output values. Since it does not drive any signal, it is a PASSIVE agent. Therefore it only has a monitor and no driver. Its internal components are shown in Figure 3.5.
This component only contains a monitor and a built-in analysis port named `monitored_ap`, which is the one responsible for broadcasting transactions of type `UART_out_transaction()` to the scoreboard. In this way, scoreboard collects actual values for comparing them to the expect ones.

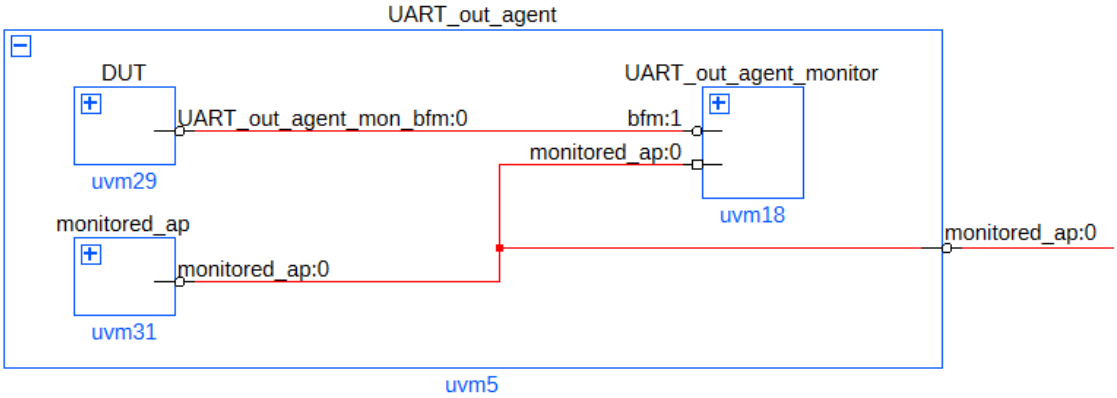

Figure 3.5: UART Agent Out from Visualizer

UART Out Agent Monitor

Since there is a dual top architecture, also the UART out agent monitor is divided in two components: a monitor class and the respective monitor BFM interface. Generator outputs the files: *UART_out_monitor.svh* and *UART_out_monitor_bfm.sv*.

UART_out_monitor.svh: it defines the class `UART_out_monitor`, which receives UART_out transactions observed by the UART_monitor BFM and broadcasts them through the analysis port on the agent. As mentioned for the UART in monitor, it defines functions and task to configure and set the handle to the monitor BFM and it captures transactions for viewing them in the waveform viewer.

UART_out_monitor_bfm.svh: it defines the sv interface `UART_out_monitor_bfm`, which performs the UART_out signal monitoring. Task `do_monitor`, is the one that implement protocol monitoring so it is the one that was modified.

Listing 3.7: modified task in UART out monitor bfm

```
interface UART_out_monitor_bfm #()( UART_out_if  bus );
 task do_monitor(output UART_out_monitor_s UART_out_monitor_struct);
  if(tx_busy_flag_i == 1'b0) begin
    flag_first_transmission = 1'b1;
  end
  if (flag_last_byte == 1'b0) begin
    while(tx_busy_flag_i == 1'b0) @(posedge clk_i);
      if (flag_first_transmission == 1'b1) begin
        repeat(3) @(posedge clk_i);
        UART_out_monitor_struct.uart_tx_dout = uart_tx_dout_i;
        repeat(bit_transmission_cc-1) @(posedge clk_i);
        flag_first_transmission = 1'b0;
      end
```

```
14        else begin
15          UART_out_monitor_struct.uart_tx_dout = uart_tx_dout_i;
16          repeat(bit_transmission_cc-1) @(posedge clk_i);
17        end
18        if(tx_busy_flag_i == 1'b0) begin
19          flag_last_byte = 1'b1;
20          n_bits = 0;
21        end
22    end
23    else begin
24     // data_frame_length = start + data + parity + stop;
25     if(n_bits < data_frame_length - 1) begin
26       UART_out_monitor_struct.uart_tx_dout = uart_tx_dout_i;
27       repeat(bit_transmission_cc-1) @(posedge clk_i);
28       n_bits = n_bits +1;
29       if(n_bits == data_frame_length - 1) begin
30         flag_last_byte = 1'b0;
31         n_bits = 0;
32       end
33     end
34    end
35 endtask
36 endinterface
```

Comparing it to the one developed for the UART In Monitor, this task was more complex to implement, since it has to capture DUT output values. Due to the fact that the DUT is a UART transmitter, it receives input values, adds start bit, parity and stop bit depending on the configuration, and then outputs value one bit at a time. For this reason, the number of clock cycles required to transmit a single bit had to be calculated in order to implement the proper protocol timing. This variable is contained in `bit_transmission_cc`, which was obtained using the following formula:

$$bit\_transmission\_cc = \frac{ClockFrequency[Hz]}{BaudRate[bps]}$$

where *Clock Frequency* is the frequency of the clock, and *Baud Rate* is the value of baud rate for the UART, that can be configured in the test.
When UART tx transmits input data, it uses `tx_busy_flag` to indicate that the transmission line is busy. This signal raises to logic 1 when a transmission starts, and it goes back to logic 0 when the second last byte of the word is transferred; last byte is still transmitted even if the busy signal is at 0. Therefore, `tx_busy_flag` is utilized as a flag to indicate whether it is the first byte to be sent or the last, as well as a signal to determine when to sample values.
If it is not the final byte to be transmitted, then monitor waits for `tx_busy_flag` to go at 1 . The bit must be fully transferred, which takes `bit_transmission_cc` clock

cycles, before the new value can be sampled. Due to this a `repeat() @(posedge clk)` is used, since it allows to wait for the desired number of rising edge of the clock.
On the other hand, if it is the last byte to be sent and `tx_busy_flag` is at 0, a check on the number of bits to be transferred is required. For this reason, the configuration variable `data_frame_length` is used. It indicates the number of bits sent in a transmission (start bit + data bits + parity + stop bit), and whenever a bit is transferred, a counter advances its value up to `data_frame_length`.
This approach allowed to appropriately gather UART DUT outputs and show the UART Out Transaction in the waveform viewer.

3.4.3 UART Predictor

UART Predictor is an analysis component which implements a golden reference model of the DUT. It provides the UART Scoreboard with expected values so that they can be compared to the actual ones. It contains analysis exports for receiving data and analysis ports for sending data, as shown in Figure 3.6:

- `UART_sb_ap`: analysis port.

  It broadcasts transactions of type `UART_out_transaction()` to the UART Scoreboard.

- `UART_in_agent_ae`: analysis export.

  It receives transactions of type `UART_in_transaction()` from UART In Agent.

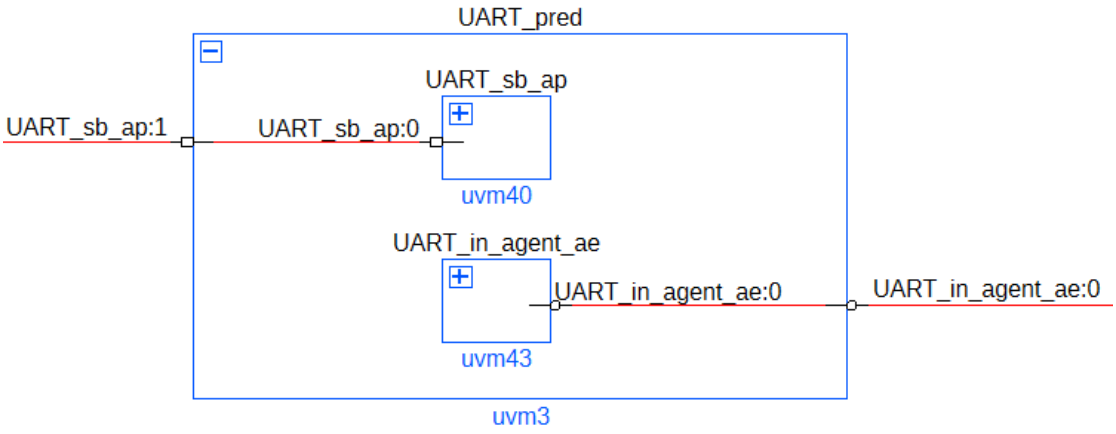

Figure 3.6: UART Predictor from Visualizer

UVMF generator creates the file *UART_predictor.svh* which defines the class `UART_Predictor`. The function that was modified in order to implement the

UART tx behavior was `write_UART_in_agent_ae()`.

Listing 3.8: modified function in UART predictor

```
class UART_predictor #() extends BASE_T;
virtual function void write_UART_in_agent_ae(UART_in_transaction() t);
  if(t.tx_num == 0) begin
    fill_memory(t);
  end
  else begin
    fill_memory(t);
    for(i = 0; i < t.tx_num; i++) begin
    if(mem[i] == 0) begin
        empty_mem_data_frame(i);
    end
    end
    // start transmitting
    for(i = 0; i < t.tx_num; i++) begin
        for(j = 0; j < data_transmitted; j++) begin
            UART_sb_ap_output_transaction =
   UART_sb_ap_output_transaction_t::type_id::create("
   UART_sb_ap_output_transaction");
            UART_sb_ap_output_transaction.uart_tx_dout = mem[i][j];
            UART_sb_ap.write(UART_sb_ap_output_transaction);
        end
    end
  end
endfunction
end class
```

This function predicts DUT output values based on DUT input, configuration, and state. It is executed when a transaction is received through `UART_in_agent_ae`. Since `tx_num` is the transaction variable that starts a transmission, different operations were carried out based on whether or not its value was equal to zero:

- `tx_num` = 0 → input data contained in the received transaction are collected. This operation is performed by the function `fill_memory()`.

- `tx_num` != 0 → input data contained in the received transaction are still collected. Next, the memory content is checked. If there are more bytes to send compared to what is written in the memory, then start, stop, and parity bits are added, according to the UART configuration, for the memory locations that contain only zeros. This operation is performed by the function `empty_mem_data_frame()`. After that, transmission of stored values is performed and output values have to be broadcast to the UART Scoreboard. Steps involved to accomplish this were: first, create a transaction of type `UART_out_transaction`; next, fill it with the appropriate value to be

sent; and finally, write the `UART_out_trasnaction` in the analysis port of the scoreboard by calling the `write()` function.

In order to store input data of `UART_in_transaction` received, a memory element was instantiated. It was implemented as a 64x12 static matrix. Thus, each of the 64 memory locations includes a vector with a maximum of twelve bits. The maximum dimension of the vector was determined by the UART setup: as UART can send eight data bits, two stop bits, a start bit, and a parity bit, the maximum bit count is twelve. In fact, each vector contains the byte that needs to be transferred along with the start, stop, and parity bits, and it represents the data that will be sent over the UART channel.
The next piece of code reports the additional custom function `fill_memory()`, which accomplishes the task of filling the memory with the right data to be communicated over the UART.

Listing 3.9: function `fill_memory` in UART predictor

```
function void fill_memory(UART_in_transaction #() t);
wr_en = 0;
i_mem = t.tx_buffer_addr * 4;
for(i = 0; i < (n_bits_word - n_bits_data + 1); i++) begin
    mem[i_mem][0] = 0; // set the startbit, it is always = 0 in UART
    j_mem = 1;
    even=0;
    if(t.tx_buffer_en_wr[wr_en] == 1) begin
        for(j = i; j < (n_bits_data + i); j++) begin
            mem[i_mem][j_mem] = t.tx_buffer_data[j];
            j_mem++;
            if(t.tx_buffer_data[j] == 1) begin
                even++;
            end
        end
    end
    else begin
        for(j = i; j < (n_bits_data + i); j++) begin
            mem[i_mem][j_mem] = 0;
            j_mem++;
        end
    end
    wr_en++;
    if(parity_en == 1) begin
        if(parity_sel == 1) begin     // ODD PARITY
            if(even % 2 == 0) begin
                parity = 1;
            end
            else begin
                parity = 0;
            end
```

```
32          end
33          else begin
34              if(even % 2 == 0) begin    // EVEN PARITY
35                  parity = 0;
36              end
37              else begin
38                  parity = 1;
39              end
40          end
41          mem[i_mem][j_mem] = parity;    //add parity bit
42          j_mem++;
43      end
44      if(stop_bit == 1) begin       //add stop bit, depending on the
        configuration
45          mem[i_mem][j_mem] = 1;
46      end
47      if(stop_bit == 2) begin
48          mem[i_mem][j_mem] = 1;
49          j_mem++;
50          mem[i_mem][j_mem] = 1;
51      end
52      i = j - 1;
53      i_mem++;
54  end
55  endfunction
```

`fill_memory()` receives as input a `UART_in_transaction`. First, it retrieves configuration variables (not reported in the code above), then it implements the algorithm to fill the memory. As this function reads `UART_in_transaction`, which comprises input data, it executes operations to load the appropriate data frame into memory for transmission over the UART:

1. Correct memory index is calculated, based on `tx_buffer_addr`.
2. Word of 32 bits is divided into bytes.
3. Start bit is added.
4. If `tx_buffer_en_wr` for the current byte is set at 1, then memory vector is filled with input data received from the transaction and the number of bits at logic 1 is determined, to compute parity. Otherwise, zeros are inserted.
5. When parity is enabled, the choice of parity bit is made based on whether parity is odd or even. Otherwise, parity bit is not included.
6. Lastly stop bits are added, depending on UART configuration.

Steps 3, 4, 5, 6 are repeated for each byte.

3.4.4 UART Scoreboard

UART Scoreboard is an analysis component which verifies that the DUT is performing as expected. As shown in Figure 3.7, it contains two analysis exports:

- `expected_analysis_export`: accessed through `write_expected` function. It receives transactions from the UART Predictor.
- `actual_analysis_export`: accessed through `write_actual` function. It receives transactions from the UART Out Agent.

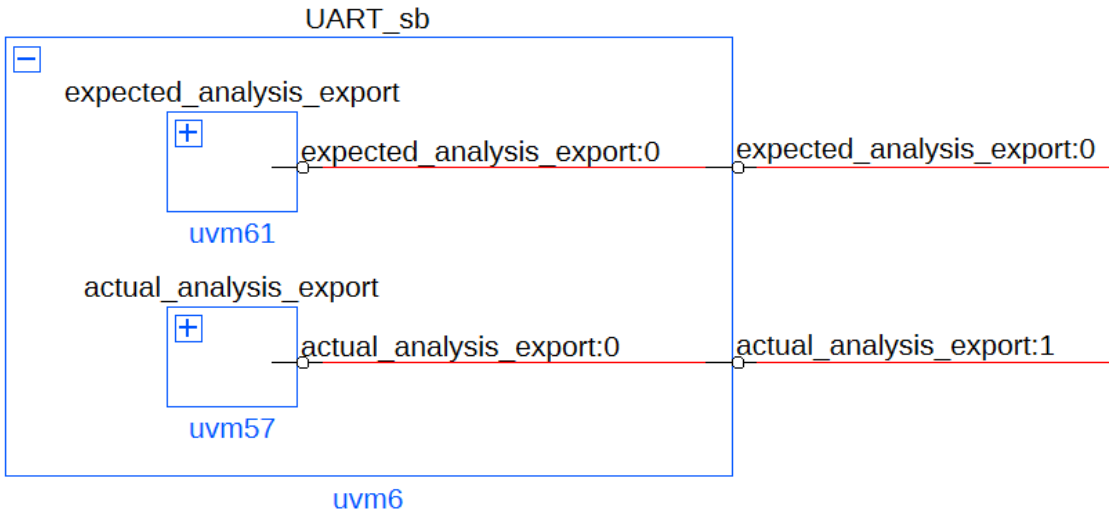

Figure 3.7: UART Scoreboard from Visualizer

UART Scoreboard operates on transactions of type `UART_out_Transaction`, and it compares them assuming an in-order arrival: expected and actual values arrive in the same order. For this reason, its class type in the *UART_environment.yaml* configuration file is declared as `uvmf_in_order_scoreboard`, which is a UVMF base library component.
`uvmf_in_order_scoreboard` is a class that defines an in-order scoreboard and extends the `uvmf_scoreboard_base` class. It includes functions for accessing the analysis ports:

- `write_expected`: it queues transactions received from UART predictor to wait for the actual ones.
- `write_actual`: it compares the transaction received from the UART Out Agent to the next transaction in the queue that contains the expected values. It displays two messages: **MATCH**, if expected and actual transaction are the same, **MISMATCH** otherwise.

When MATCH/MISMATCH are printed, the expected and actual transaction content is included as well to help the user to detect possible errors.
`UART_Scoreboard` class, which extends *uvmf_in_order_scoreboard* class, was implemented to add some custom functions to the base scoreboard generated by UVMF. Added functions are reported in the following code.

Listing 3.10: UART Scoreboard custom functions

```
class UART_scoreboard #(type T = uvmf_transaction_base, type BASE_T =
    uvmf_in_order_scoreboard#(T)) extends BASE_T;
virtual task run_phase(uvm_phase phase);
  UVM_FILE scoreboard_log_fh = $fopen("scoreboard_output.log");
  super.run_phase(phase);
  set_report_id_action("SCBD", UVM_LOG | UVM_DISPLAY);
  set_report_id_file("SCBD", scoreboard_log_fh);
endtask

virtual function void report_phase(uvm_phase phase);
  UVM_FILE scoreboard_log_fh = $fopen("scoreboard_output.log");
  super.report_phase(phase);
  if(mismatch_count == 0) begin
   `uvm_info("SCBD", $sformatf("TEST PASSED"), UVM_LOW)
  end
  else begin
   `uvm_error("SCBD", $sformatf("TEST FAILED"))
  end
  $fclose(scoreboard_log_fh);
endfunction
endclass
```

The `uvmf_scoreboard_base` class already included the functions `run_phase` and `report_phase`, which are defined for each UVM component. However, exploiting the object-oriented programming potential, they were overridden in the class `UART_Scoreboard` to provide additional operations.

- `run_phase`: opening of a file is included. UART Scoreboard must also print results of the comparison to an output file, in addition to showing them in the QuestaSim console. For this reason, the types of messages that have to be printed in the log file are also specified, using the id type. Since all messages from the scoreboard have to be printed the id type is set to `"SCBD"`.

- `report_phase`: it is used to gather results at the end of the comparison. In this instance, UART scoreboard writes **TEST PASSED** if it detected zero mismatch, and **TEST FAILED** otherwise. This messages are printed both on the console and on the output file. At the end of this function, the file is correctly closed.

Two types of UVM messages were used for this testbench: `uvm_info` for displaying 'MATCH' and 'TEST PASSED', and `uvm_error` for printing 'MISMATCH' and 'TEST FAILED'.
The output file was named *scoreboard_output.log*, and it shows all the messages with the identifier `SCBD`, which represent all the messages printed by the UART Scoreboard class.
Since uvm messages include also the path of the file and the simulation time in which the message is printed, these details were also printed in the output file. For this reason, a python script (`modify_output_file.py`) was implemented in order to modify *scoreboard_output.log* and remove the additional information. The resulting output file was named *test_output.txt* and reports only messages about a MATCH/MISMATCH, the number of predicted transactions and the string "TEST PASSED" (or "TEST FAILED").
Last lines of these two files are reported in Figure 3.8 and Figure 3.9, as example:

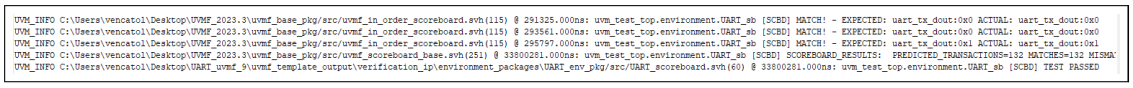

Figure 3.8: scoreboard_output.log example

```
MATCH! - EXPECTED: uart_tx_dout:0x0 ACTUAL: uart_tx_dout:0x0
MATCH! - EXPECTED: uart_tx_dout:0x0 ACTUAL: uart_tx_dout:0x0
MATCH! - EXPECTED: uart_tx_dout:0x1 ACTUAL: uart_tx_dout:0x1
SCOREBOARD_RESULTS:  PREDICTED_TRANSACTIONS=132 MATCHES=132 MISMATCHES=0
TEST PASSED
```

Figure 3.9: test_output.txt example

It can be seen that the output messages printed in *scoreboard_output.log* file are not easily readable since there is a lot of information, which is not required. On the other hand, *test_output.txt* file reports only important information about the test, so: if a transaction produces a match/mismatch, a final count and the string TEST PASSED.

Chapter 4

Results

Following the proper development of the UVM testbench for the UART transmitter, tests were run to verify that the testbench worked as expected.
Tests were run using the sequences outlined in the previous section, and changing only the UART configuration to check different combinations of baud rate, parity, and stop bits. `test_top` class receives the UART configuration from a text file, where the user can enter the desired value in a binary format.
An example of simulation to show that the testbench functioned as expected is reported below. All the waveform images were taken from the simulation run in QuestaSim.
Finally, advantages and disadvantages of developing a testbench following a UVM approach are discussed.

Test example: UART configuration = 11101000

Selecting a configuration of "11101000" (hex = e8) indicates that the testbench variable `uart_configuration` has been configured with the values reported in Table 4.1:

Bit	Value	Description
7	1	UART transmitter enabled.
6	1	Parity odd
5	1	Parity enabled.
4	0	One stop bit
3:0	1000	Baud rate = 460 Kb/s.

Table 4.1: UART configuration (e8) register details

With this input configuration a data frame of eleven bits is transmitted: 1 start bit, 8 data bits, 1 parity bit and 1 stop bit.

In the following, `UART_in_agent` and `UART_out_agent` waveform are shown, in order to see the correct testbench development.

`UART_in_agent` waveform: they are showed in Figure 4.1.

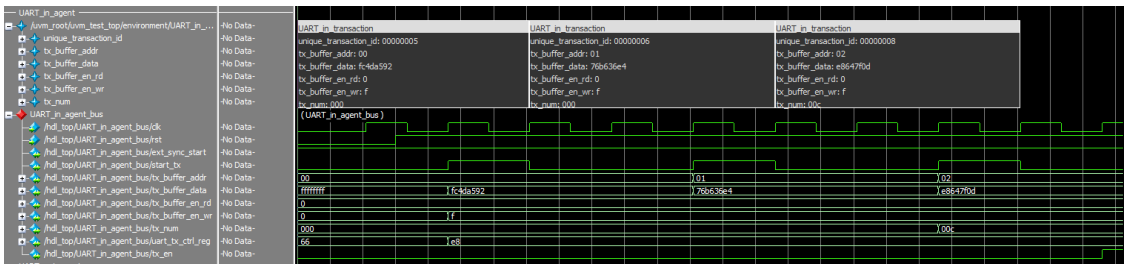

Figure 4.1: UART In Agent waveform uart_config = e8

Firstly, `UART in transactions` are showed:

- First transaction: `tx_num` is equal to '0', so it only load this data in the memory. `tx_buffer_data` contains a random value that is written in `tx_buffer_addr = 0` memory address.
- Second transaction: `tx_num` is equal to '0', so it only load this data in the memory. `tx_buffer_data` contains a random value that is written in `tx_buffer_addr = 1` memory address.
- Last transaction: `tx_num` is equal to '1', so it load this data in the memory and start a transmission. `tx_buffer_data` contains a random value that is written in `tx_buffer_addr = 2` memory address.

First two transactions are created by the `UART_in_fill_sequence` class, while the last one is created by `UART_in_start_tx_sequence`. For all these transactions, `tx_en_wr` is equal to "1111", so all bytes are transmitted over the UART and the variable `tx_buffer_data` contains a random value.
After the last transaction, collected data are transmitted to the DUT, and this can be checked by looking at the `tx_en` signal that changes its value to logic '1'.
So, UART In Agent behaves as expected.

`UART_out_agent` waveform: in the next figures, several screenshots of waveform are reported, due to the fact that the `UART_out_transaction` represents a single output bit. As there are many transactions, they have been split for easier viewing.
Figure 4.2 shows how a single output transaction appears in the simulation viewer. It lasts `bit_transmission_cc` clock cycles, that are the time needed to correctly transmit the bit over the uart with the selected configuration.

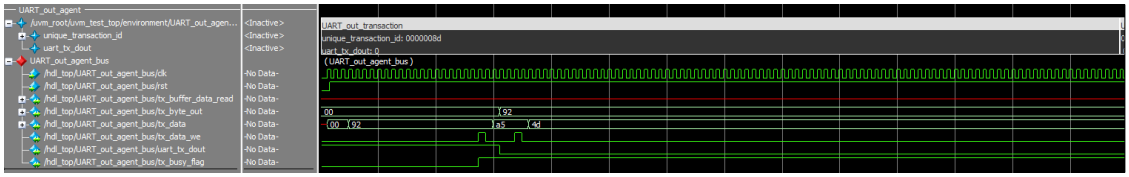

Figure 4.2: UART Out Agent Single transaction uart_config = e8

Figure 4.3 shows the output transaction corresponding to the first input transaction that was sent. The data transmitted is `tx_buffer_data = fc4da592`, and it is transmitted over the UART one byte at a time. Output transactions must be evaluated in 11-bit groups, with each representing a byte plus the start, stop, and parity bits. By comparing the first transactions group with the first byte (expressed in hexadecimal) displayed by `tx_byte_out`, it is possible to verify that the output transactions are collecting the correct DUT output values.
Same consideration can be done for the other three bytes transmitted, to complete the transfer of the first input transaction.

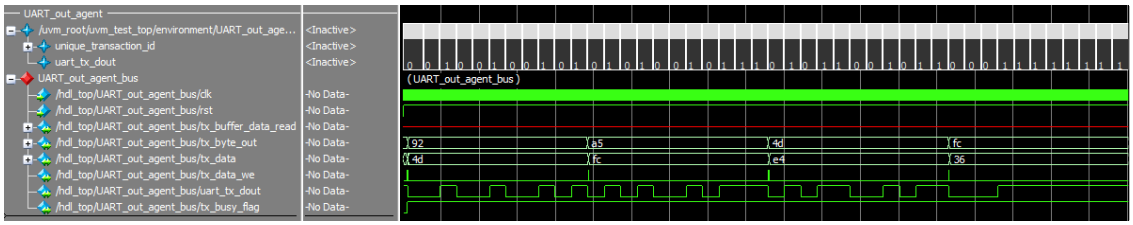

Figure 4.3: UART Out Agent First transaction uart_config = e8

Figure 4.4 and Figure 4.5 shows transactions corresponding to the second and the third input transactions. By examining their waveform, it is possible to determine whether or not each byte was transmitted correctly, as it was done for the first output transaction.

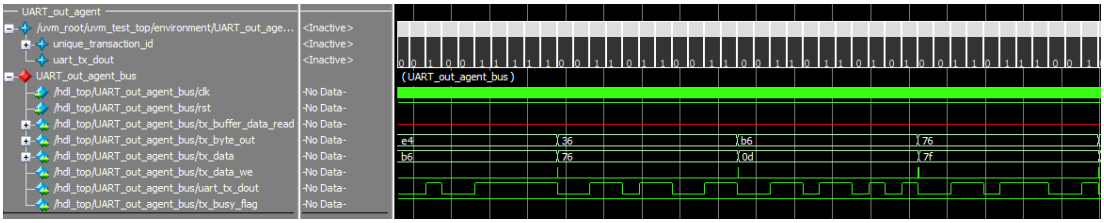

Figure 4.4: UART Out Agent Second transaction uart_config = e8

Figure 4.5: UART Out Agent Last transaction uart_config = e8

UVM messages printed in QuestaSim console, also helped to identify if the output transaction, which collect actual DUT outputs, matched the golden output value. In fact scoreboard prints all the transaction it has received and print MATCH/MISMATCH accordingly.
In conclusion, these images along with the printed UVM messages, show that each UVM component operates in the correct way. In the following UVM messages for the last two output transactions and end of test messages are showed:

```
1 UVM_INFO @ 282271.000ns: uvm_test_top.environment.UART_sb [SCBD]
    MATCH! - EXPECTED: uart_tx_dout:0x1 ACTUAL: uart_tx_dout:0x1
2 UVM_INFO @ 282271.000ns: uvm_test_top.environment.UART_out_agent.
    UART_out_agent_monitor [MON] uart_tx_dout:0x1
3
4 UVM_INFO @ 284421.000ns: uvm_test_top.environment.UART_sb [SCBD]
    MATCH! - EXPECTED: uart_tx_dout:0x1 ACTUAL: uart_tx_dout:0x1
5 UVM_INFO @ 284421.000ns: uvm_test_top.environment.UART_out_agent.
    UART_out_agent_monitor [MON] uart_tx_dout:0x1
6
7 UVM_INFO @ 32500271.000ns: reporter [TEST_DONE] 'run' phase is ready
    to proceed to the 'extract' phase
8 UVM_INFO @ 32500271.000ns: uvm_test_top.environment.UART_sb [SCBD]
    SCOREBOARD_RESULTS: PREDICTED_TRANSACTIONS=132 MATCHES=132
    MISMATCHES=0
9 UVM_INFO @ 32500271.000ns: uvm_test_top.environment.UART_sb [SCBD]
    TEST PASSED
```

Additional tests were run to verify different UART configurations, and each of them was successful.

Evaluations

The developed UVM environment for testing the UART transmitter device performed a punctual test of the DUT, meaning that every bit of the serial line was tested. Since there is not a VHDL-based testbench that was previously created and run the same test, it is not possible to have performance parameters to compare.

However, studying and developing a UVM testbench allows for some consideration about potential benefits and drawbacks of this approach in the context of hardware verification conducted in Leonardo Electronics.
Advantages that a UVM approach could bring are:

- **High abstraction level**: UVM uses a higher abstraction level if compared to a VHDL-based testbench. As a result, the code is simpler to understand and maintain, allowing a test engineering to concentrate more on the functionalities and behavior of the design rather than on the low level details.
- **Hierarchical structure**: UVM provides hierarchical structure that makes easier to manage and organize the testbench complexities.
- **OOP features**: OOP allows for the creation of modular and reusable UVM components. In fact, OOP allows to create base classes, that defines common functionalities, and these can be extended to create specific instances for different interfaces. As a result, child classes can inherit methods and properties from parent classes through inheritance. Additionally, objects of different classes can be considered as objects of a single common class thanks to polymorphism, which is helpful when developing generic test sequences that can run on various interfaces or configurations.

Furthermore, the use of UVM Framework speeds up testbench development by automating component instantiation and connection. This process makes it easier to understand how UVM works and allows engineers to focus on adding DUT specific functionalities.
On the other hand, one of the drawbacks that was brought to light throughout this procedure is that not all the UVM components in the testbench can be synthesized, but only the BFMs. This is a flaw in the Leonardo Electronics testing process since it is crucial that the electronic device correctly works in both actual hardware and simulation. In fact, in the current company verification flow, the developed VHDL-based testbench can be used as it is both in simulation and in real hardware. Since a UVM-based testbench cannot be used in the same way, research into how to include it into emulation is necessary.

Chapter 5

Conclusion

This thesis work conducted an extensive study and exploration of the Universal Verification Methodology (UVM) in order to investigate the potential of this verification approach. After some preliminary study, a UVM testbench was carefully developed for a UART transmitter device.
First the necessary UVM components were selected, and then configuration files were produced using the YAML language to describe them. These files were needed as input for the UVMF code generator. This methodical approach, allowed the start of development within the UVMF framework, creating a solid basis for subsequent changes. Particularly, the UVMF code generator eased the instantiation and interconnection of required UVM components, which sped up the development process. Starting from the UVMF-generated testbench, it was possible to begin characterizing each UVM component in order to meet DUT functional requirements. The developed UVM Environment included two agents, one for the input interface and the other for the output interface, a predictor, and a scoreboard. The agents' monitor and driver BFMs were customized to ensure proper communication with the DUT. The predictor reproduces the DUT behaviour in order to compute the golden reference value to compare to the actual DUT outputs, for this reason a proper analysis of the UART transmitter behavior was performed. The scoreboard compares the actual values with the golden ones and outputs messages to the console and an output file. Sequences and transactions were also customized to provide stimulus to the environment: three sequences were added to perform correct testing. Simulations run in QuestaSim were fundamental to prove the proper development of each component. Correct UVM components characterization was challenging, however it proved essential in improving understanding of UVM functionalities.
Finally, the completed development of a comprehensive testbench represented a significant achievement, making it possible to examine the potential advantages of the UVM verification process over the VHDL-based testbench. Higher abstraction layer and modular architecture of UVM simplifies the testbench development

process, allowing test engineering to focus on behavioral functionalities rather than complicated low level ones.
The architecture for the developed UVM environment provides a starting point for future improvements, which may include the addition of new test sequences or the improvement and extension of components, such as a more customized scoreboard. Furthermore, the extension of this testbench enables testing of a DUT made of both a UART transmitter and the equivalent UART receiver, increasing its applicability. Future studies will also focus on integrating Siemens EDA's Questa Verification IP (QVIP) into a UVM testbench. In fact, QVIP is a library of verification IP solutions that supports several industry-standard protocols and ensures flexibility for design verification.
In conclusion, this thesis activity provided hands-on experience with verification frameworks, resulting in excellent learning opportunities. In particular, it provided an opportunity for gaining competence in System Verilog, a hardware verification language, and also becoming familiar with UVM, a widely used methodology in the verification context.

Bibliography

[1] National Instruments. *What is DO-254?* URL: https://www.ni.com/en/solutions/aerospace-defense/what-is-do-254-.html (cit. on p. 1).

[2] Siemens EDA. *Verification Academy*. URL: https://verificationacademy.com/ (cit. on p. 3).

[3] Slides Siemend EDA. *SystemVerilog for Verification, Student Workbook* (cit. on p. 4).

[4] A. Fiergolski. «Simulation environment based on the Universal Verification Methodology». In: (Jan. 2017) (cit. on p. 5).

[5] Siemens EDA Verification academy. *UVM Cookbook* (cit. on pp. 5–7, 9).

[6] Slides Siemens EDA. *SystemVerilog UVM, Student Workbook* (cit. on pp. 5, 8).

[7] Chip Veirify. *UVM Tutorial*. URL: https://www.chipverify.com/uvm/uvm-tutorial (cit. on pp. 7, 8).

[8] Verification Academy. *UVM Framework*. URL: https://verificationacademy.com/topics/uvm-universal-verification-methodology/uvmf/uvm-framework/ (cit. on p. 10).

[9] Siemens EDA. *UVM Framework Users Guide*. Version 2023.3 (cit. on p. 10).

[10] Siemens EDA. *UVMF YAML Reference Manual*. Version 2023.3 (cit. on pp. 12, 17).

[11] Mary Grace Legaspi Eric Peña. «UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter». In: (Dec. 2020) (cit. on p. 13).