Implementation of an OVM Based Advanced Functional Testbench for a Mixed-Signal A/D Converter

Student: JULIEN GHAYE  
Supervisor: ALAIN VACHOUX  
Microelectronic Systems Laboratory  
Swiss Federal Institute of Technology, Lausanne

Master’s thesis  
June 2009
Acknowledgements

I would like to thank Alain Vachoux who accepted me to work on this project at LSM. I wish also to thank Thomas Brücker for having proposed this subject to LSM and all the staff members from Semetch for their every day support and advices, for their help through all steps of my work.

Finally, I would like to thank Marie-Jane Savary, my landlady, to put up with my completely uncommon working hours. And of course, I wish to thank my family which is in Belgium and all my friends for their every day support and encouragements.
Master Diploma Project 2009

Implementation of an OVM Based Advanced Functional Testbench for a Mixed-Signal A/D Converter

Student: Julien Ghaye
Responsibles: Alain Vachoux (EPFL)
Joachim da Costa (Semtech)
Thomas Brückner (Mentor Graphics)
Duration: February 16, 2009 – June 19, 2009

Introduction and Motivation

Today’s systems-on-chip (SoC) can be multi-million gate ASICs and FPGAs including complex components (e.g., processor cores, large memories, codecs, etc.) and running megabytes of embedded software (e.g., firmware, (real-time) operating system, application software, etc.). As the design of such systems grows in complexity, the need for supporting concurrent functional verification becomes increasingly prominent to develop the right product in optimum time. One key issue is then to include verification tasks as early as possible in the design process. It is widely recognized today that verification can consume up to 70% of the total design effort and that the number of verification engineers can be up to twice the number of design engineers in a particular design project.

New languages, tools and methodologies recently emerged as efficient solutions for addressing functional verification of SoCs. The SystemVerilog language is an extension of the Verilog language that provides a powerful set of constructs for developing fairly sophisticated verification environments, while still maintaining links to the traditional RTL design. The SystemVerilog language is the IEEE standard 1800 since 2005, and all major EDA tools have been updated to support it. The Open Verification Methodology (OVM, http://www.ovmworld.org) is an open-source, SystemVerilog-based class library that eases the development of modular, multi-layer and reusable verification environments.

Semtech – a leading mixed-signal IC design house based in Neuchâtel, Switzerland – developed a medium-complexity mixed-signal zooming A/D converter with a small custom 12-bit processor and an I2C interface. It is a data acquisition system based on the ZoomingADC™ technology. It directly connects most types of miniature sensors with a general purpose microcontroller. So far, the design and the verification have been done using the VHDL language. However, this approach reached its limits with the complexity of the design and of the stimulus that are required to test it. Higher modeling abstraction levels, better means to express specifications and properties, and better means to measure performances are required. Also, the verification of mixed-signal designs needs to verify continuous-time properties that cannot be easily specified.
**Project Objectives**

The goal of the project is to develop a functional verification environment for the Semtech’s data acquisition system based on the ZoomingADC™ technology [SX8724]. The verification environment will be implemented using the SystemVerilog language [Suth06][Spear06] and will be compliant to the OVM 2.0 (Open Verification Methodology) specification [OVMREF][OVMUG]. The verification environment will be validated by using the Questa advanced verification platform from Mentor Graphics [QuestaDS].

Existing VHDL testbenches will be replaced with an OVM class-based environment that will define a new verification environment targeted at complex mixed-signal systems. The new testbench environment shall heavily leverage object oriented programming techniques available in SystemVerilog and AFV (Advanced Functional Verification) methods such as SystemVerilog assertions (SVA), functional coverage, constraint random stimuli and testbench automation. The new approach will then be compared to the previous one using VHDL-based testbenches.

A report will summarize the work done, the key strengths of the OVM approach and the results achieved with the new verification methodology.

**Work Organization**

The work will be done partly in EPFL and partly in Semtech Neuchâtel. For confidentiality reasons, the design databases will remain located on Semtech servers. The work requesting access to the databases must therefore be executed in Semtech’s facilities in Neuchâtel. The remaining of the work can be done at the EPFL.

The following topics will have to be mastered in the project:

- The use of the SystemVerilog language for design and verification.
- The use of the Questa AFV tool from Mentor Graphics.
- The use of the OVM class library.
- The development of a functional verification environment and its associated methodology.
- The functionalities of the SX8724 circuit from Semtech.
- The VHDL testbench environment of the SX8724 circuit.

The planning is TBD.

**References**


# Contents

## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Motivation .................................. 1
1.2 Thesis organisation .......................... 2
1.3 The Design Under Test ....................... 3

## 2 Verification concepts

2.1 General concepts .............................. 9
2.1.1 Directed and constrained random tests .... 10
2.1.2 Scoreboarding .............................. 12
2.1.3 Functional coverage ....................... 13
2.1.4 Layered testbench .......................... 13
2.1.5 Transaction level modeling ................. 14

2.2 OVM key concepts .............................. 18
2.2.1 OVM mechanics .............................. 18
2.2.1.1 OVM components ......................... 18
2.2.1.2 The OVM factory ......................... 20
2.2.1.3 Configuration of OVM components ....... 21

2.2.2 Virtual interfaces .......................... 22
2.2.3 Tests and stimulus generation ............... 24
2.2.3.1 Sequences .............................. 24
2.2.3.2 Tests .................................. 25
3 SX8723 tests and testbenches

3.1 VHDL testbench ...................................................................................................... 27
3.2 OVM testbench ........................................................................................................ 30
  3.2.1 Global architecture ............................................................................................... 30
  3.2.2 DUT wrappers ...................................................................................................... 34
  3.2.3 SystemVerilog interfaces ...................................................................................... 38
    3.2.3.1 Chip interface .................................................................................................. 38
    3.2.3.2 I2C interface ................................................................................................. 41
  3.2.4 Transactions .......................................................................................................... 43
    3.2.4.1 I2C transactions ............................................................................................ 44
    3.2.4.2 Mixed signal transactions ............................................................................ 48
  3.2.5 OVM components ................................................................................................ 50
    3.2.5.1 I2C agent ....................................................................................................... 50
    3.2.5.2 I2C analysis component ............................................................................... 53
    3.2.5.3 Mixed signal agent ....................................................................................... 57
    3.2.5.4 Mixed signal analysis component ................................................................. 62
  3.2.6 Using the testbenches ............................................................................................ 66
    3.2.6.1 Library of sequences ..................................................................................... 66
    3.2.6.2 Tests ............................................................................................................... 69

4 Further discussions ....................................................................................................... 73

4.1 Simulation including timings ..................................................................................... 73
4.2 Reusability of the developed components ............................................................... 75
4.3 What about assertion based verification? ................................................................. 76

5 Conclusions ................................................................................................................. 79

5.1 Comparison between the old VHDL and the new OVM testbench ......................... 79
5.2 Final note .................................................................................................................. 81

A Selected code source .................................................................................................. 82

A.1 Top level module ...................................................................................................... 82
A.2 VHDL wrapper ........................................................................................................... 84
# List of Figures

1.1 SX8723 Block Diagram, display also the different pins and their associated type ........................................ 4  
1.2 ZoomingADC Block Diagram .......................................................... 5  
1.3 I^2C communication format ............................................................ 5  

2.1 Separation between tests and testbenches ................................. 10  
2.2 Functional coverage over time ........................................................ 11  
2.3 Scoreboarding concept ................................................................. 12  
2.4 Different types of TLM connections ............................................... 16  
2.5 Different types of TLM connections featuring FIFOs channels ........ 17  
2.6 Tree hierarchy of the OVM components ......................................... 18  

3.1 Global architecture of the VHDL testbench formerly developed by Semtech. 28  
3.2 Global architecture of the new OVM testbench ............................ 32  
3.3 Internal design of the different wrappers ...................................... 35  
3.4 Internals of the chip_if SystemVerilog interface ............................ 39  
3.5 Internals of the i2c_if SystemVerilog interface .............................. 42  
3.6 I^2C agent as implemented in the default environment .................. 50  
3.7 I^2C analysis component as implemented in the default environment ... 54  
3.8 PIO, DIG and AC agent as implemented in the default environment .... 58  
3.9 I^2C PIO, DIG and AC analysis component as implemented in the default environment .................................................. 63  

4.1 Voltage change on a single port of the DUT ............................... 74  

B.1 Waveform of the ZoomingADC™ in action ............................... 105
B.2 Functional coverage after the execution of a constrained random test
  using Questasim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
Chapter 1

Introduction

1.1 Motivation

The introduction of hardware description languages (HDL) like Verilog or VHDL in the late 1980s allowed electronic engineers to design digital electronic circuits at a higher level of abstraction than the widely use switch and gate level. This new level of abstraction along with the fast evolution of electronic design automation (EDA) tools and computing power throughout the years give today’s engineers the ability to design highly complex systems.

Along with the design of the components themselves comes the need for verification. Are they working properly, according to the specifications? Every RTL model intended for synthesis has to be validated through simulation in order to go to the next step in the design flow. This validation is done by testbenches, pieces of software stimulating the RTL models and checking their responses. Those testbenches are tools developed with the design team, helping in finding bugs in the device, and, ideally, to withdraw them all. The verification is a very important part of the design flow since a bug going through synthesis, or worst through tape-out, can costs a lot of time, money and human resources.

It is quite obvious that for low complexity design, like a 4 bit adder, the verification process is quite straightforward and the RTL model can be exhaustively tested within a few hours using simple verification techniques. However, today’s multi-million gates systems-on-chip (SoCs), featuring mixed signal components, highly complex state machines, processors, embedded softwares, signal processing, ... require much more in-
involved verification techniques and methodologies to reduce as much as possible the amount of bugs in the final product.

The Neuchâtel office of Semtech\textsuperscript{1} developed a medium-complexity data acquisition system, named SX8723. This chip features an configurable multi-inputs A/D converter\textsuperscript{2} with a small 12 bits RISC processor implementing an I\textsuperscript{2}C interface. A VHDL model of the complete chip die is available. It includes on one side the different RTL models of the digital parts, and on the other side, models for the analog part. Semtech’s engineers are using a non standard VHDL package allowing them to model the analog behavior of electronic components. Thus, they have access to a complete mixed-signal VHDL model allowing them to validate the whole behavior of the device instead of testing each digital component independently.

The model verification has been done using a testbench coded in VHDL. So far so good, but this approach reached its limit quickly regarding the complexity of the chip and the fact that VHDL is not designed for verification.

The goal of this project is to implement a completely new verification architecture using new techniques.

- The target language is SystemVerilog, a set of extensions to the Verilog Hardware description language, including new features designed for verification (assertions, functional coverage, ...), object oriented concepts, and more.

- The Open Verification Methodology (OVM) is used to implement the new functional verification environment. This is an open source SystemVerilog library developed by Mentor Graphics and Cadence implementing the Transaction Level Modeling (TLM) concept, generic classes to ease the creation of testbenches and tools for stimulus generation. OVM implements much more concepts, those used in this project will be explained in time.

1.2 Thesis organisation

This document is organized in 5 different chapters.

\textsuperscript{1}Semtech Corporation, a leading supplier of analog and mixed-signal semiconductor products.

\textsuperscript{2}ZoomingADC\textsuperscript{TM} technology
Chapter 1, Introduction This is the actual chapter, introducing the work, describing the main motivation. It also makes an introduction about the chip that is being tested by the newly developed testbench.

Chapter 2, Verification concepts The verification world is a bit apart from the design world in the microelectronic field. This chapter introduces the main concepts of verification used in this project, giving the reader the basic knowledge to understand the following chapters. It however does not replace practicing nor reading more involved book if you want to go one step further and experiment the designed OVM testbench.

Chapter 3, SX8723 tests and testbenches This is the core of the project. It describes the implementation of the OVM testbench, shows the different problems that arose and the solutions that have been developed to overcome them. Prior to this, a small introduction of Semtech’s former VHDL testbench is done, giving the user the information to compare both previous and recent approach.

Chapter 4, Discussions Developing a complete architecture to test a complex system takes time. Since the time allowed for this project was limited, some points have not been investigated. This chapter simply take a few interesting cases and discuss them.

Chapter 5, Conclusions The final chapter concludes the work that has been done.

1.3 The Design Under Test

Let’s now focus a bit on the device under test (DUT) to highlight its main features and particularities. It might be a good idea to keeps the following points in mind while reading the rest of this document because the designed testbench is coded to operate and test those features. If you want to have more details about the DUT, you can refer to the official datasheet or the design specification.

The DUT, also called SX8723 or SX8724 (reduced number of analog inputs), is a data acquisition system intended for low speed sensors. The figure 1.1 depicts the bloc
1.3 The Design Under Test

Figure 1.1: SX8723 Block Diagram, display also the different pins and their associated type

diagram of the system. As it shows, the chip has two main part: the ZoomingADC™ taking care of converting an input voltage to a digital value and a RISC microcontroller.

**ZoomingADC™** This device is a highly configurable and versatile low-power analog to digital converter. Without going into the details (check the datasheets if you are looking for them), the key features of this analog front-end are:

- 4 independent differential channels, or 7 differential channels sharing a common input.
- Programmable A/D converter resolution from 6 to 16 bits.
- 3 Programmable Gain Amplifier (PGAs) providing a differential gain tunable from 1/12 to 1000, and 2 offset compensators as depicted in the figure 1.2.
- Programmable sampling frequency, from 62.5kHz to 500kHz.

**Microcontroller** This is a CoolRISC 12 bits microcontroller whose main purpose is to interface a bank of registers with the outside world via an I²C bus. The chip user need to access those registers in a read mode to configure the ZoomingADC™, to trigger some conversions and in read mode to retrieve the converted digital value. On the overall, their are 24 registers accessible via the I²C bus.
1.3 The Design Under Test

The I²C specification defines how some chips can communicate using such bus, but does not define the meaning of the data sent (beside the slave address). The figure 1.3 shows the communication format used by the DUT to interact on the I²C bus.

Read format

<table>
<thead>
<tr>
<th>S</th>
<th>Slave address</th>
<th>W</th>
<th>A</th>
<th>Register address</th>
<th>A</th>
<th>S</th>
<th>Slave address</th>
<th>R</th>
<th>A</th>
<th>Data</th>
<th>( \overline{R} )</th>
<th>P</th>
</tr>
</thead>
</table>

Write format

<table>
<thead>
<tr>
<th>S</th>
<th>Slave address</th>
<th>W</th>
<th>A</th>
<th>Register address</th>
<th>A</th>
<th>S</th>
<th>Slave address</th>
<th>W</th>
<th>A</th>
<th>Data</th>
<th>A</th>
<th>P</th>
</tr>
</thead>
</table>

Figure 1.3: I²C communication format

Let’s now take a closer look at the VHDL ports of the DUT. As we can see on the following code snippet, all the ports are of type analog, a non standard VHDL type. Even the ports that only require to vehicle digital values (pad_d2 and pad_d3 for example) are using it.

```vhdl
entity sx8723 is
  port (  
    pad_ac2 : inout analog;
    pad_ac3 : inout analog;
    pad_ac4 : inout analog;
```
The fact is this analog type is part of a complete VHDL package which purpose
is to give tools to designers for modeling analog behaviors. Thus, they are finally
able to model a complete mixed-signal chip ready for system-level verification using
only VHDL. The main advantage of the package resides in the simulation time. Their
is no more needs to rely only on low level Spice models nor use another simulator
implementing AMS features (VHDL-AMS for example).

The following piece of code comes from the analog package itself and shows the
basic structure of the analog type.

```vhdl
subtype analog is resolve_d uanalog_device;
```

The analog type is defined as a VHDL record and is associated with a resolve
function. The main goal of this type is to model an electrical node as known in the
circuit theory. As a small reminder, a node might be connected to a given number
of voltage or current generators (nb_gene) and to a given number or loads (nb_load).
Calling those node 'drivers', the VHDL resolve function is able to compute the different
characteristics of the node: its voltage ($V_{\text{value}}$) and impedance ($Z_{\text{value}}$). The sum of all currents flowing through the loads is also determined ($I_{\text{value}}$). Along with the analog type come a number of procedures defining a common API to operate variables or signals using it. We can find procedures like $v_{\text{gene}}()$ allowing the user to connect a new driver to a node, or $\text{high}_z()$ used to put a node in a high impedance state. There are also convenient procedures performing conversions with the IEEE std_logic type, used to interface analog models with RTL ones.

To sum up this introduction about the DUT, we can already point out that test-benches targeting this design are required to be able to drive and probe analog voltages on specified pins, whiles other pins will require a complete digital communication protocol like I²C. We event have completely hybrid pins, like the $\text{pad}_d0$ which, according to the configuration written within the chip registers, can act either as a digital input, as digital output or as an analog input.

Another challenge that must be mastered to achieve the goal of the project is to be able to operate, from a SystemVerilog program, non standard, non digital VHDL types.
Chapter 2

Verification concepts

Creating a new product usually involve two teams. First the designers, whose objectives are to create a device that meet the product specifications, and the verification engineers whose job is to exercise the device created by the designers. The latter are given the job to make sure the created device performs well for what it has been created, according to the specifications. Finding bugs is only a consequence of this process. Ultimately, if the verification team can proof that the new device features has been tested enough, without finding anymore bugs, the device is validated and ready to go to the next development step.

2.1 General concepts

The first, yet very important principle to introduce the separation between tests and testbenches. Verifying a design starts with an interpretation of the specifications out of which a verification plan is created. This plan defines the global strategy to test the RTL code of the designers, thus defining the different tests to run. On the other hand, a tool is required to run those tests: the testbench. The goal of the testbench is to simulate the final environment of the DUT and provide means, hooks to stimulate it.

This approach is depicted in the figure 2.1 showing the clear separation between tests and testbenches. While the testbench is a piece of code specific to the DUT, the tests are specific to the testbench itself, not the DUT directly.
2.1 General concepts

2.1.1 Directed and constrained random tests

Direct testing is the traditional approach to test the features of a design. Using the specifications, one can identify a list of features to be tested and their associated tests. This type of approach can be time consuming for big designs. We indeed, need for each test, to craft stimulus vectors to be injected in the DUT using the testbench. Afterwards, we need to manually review the log files to check if the test performed well. If so, we jump to the next step in the design plan until all the features listed features have been covered. The solid graph in the figure 2.2 shows the coverage evolution of such a verification plan over the time. If we have enough time, we can expect to have covered all the features of the design, assuming the verification gathers them all!

To avoid having to generate by hand all the stimulus vectors required to test a design, we can write tests that generate them randomly. The time we were spending writing directed tests is transformed into CPU time, the simulator taking care of generating the stimulus that will exercise the design features. However, it seems obvious that pure randomly generated stimulus will lead us nowhere. The random generator might take way too much time to generate useful stimulus vectors. This why each test using randomization should be associated with constraints giving the simulator guidelines to generate useful stimulus vectors. This is what we call constrained random tests.

Randomization has to be done wisely. Too broad constraints lead to an increased
2.1 General concepts

A question that might arise is *What can be randomized?*. The answer is merely everything. If the testbench has been carefully developed, we can start randomizing a lot of different aspects:

- The environment can be randomized. If the testbench allow the user to configure the DUT environment, we can generate it randomly putting the DUT in different situations. This can be useful since we cannot know how the final customer will use the device we are creating and with which kind of other components it might be interfaced.

- The configuration of the DUT might also be randomized during the simulation, thus randomizing the features tested by a single test.

- The input data can be randomized as well.

- Errors injection and generation can be randomized and mixed with other error free stimulus vectors, testing the reaction of the DUT while facing protocol or timing errors.

simulation time, while too narrow ones are making these tests look like directed ones, loosing the advantages of randomization. Thus, well chosen constraints allow the simulator to test design features faster than using a directed approach. The dashed graph on the figure 2.2 shows that evolution.
Unlike the directed tests, constrained random tests are not focused on a particular feature but on many features at the same time. One of the great advantages of the randomization is to be able to generate stimulus vectors and to put the DUT in a state that might have not been specified by the verification plan, revealing unknown states of the DUT that might contain bugs. Those ones were, for example, impossible to detect using the directed approach. Finally, there will always be corner cases that randomization cannot test easily where directed tests are still much more efficient.

2.1.2 Scoreboarding

When generating randomized stimulus vectors, we need the testbench to be able to detect if the DUT response is as it should be. When only directed tests are used, the expected responses are pre-computed or manually checked because the stimulus vectors are hand-written. Now that we don’t know them before the simulator generates them, we need to have some mechanisms to generate expected responses on-the-fly.

This is the job of a high-level, software model of the DUT called the reference or golden model. Randomly generated stimulus are sent both to the DUT and that reference model, each generating a response that must be compared by a scoreboard. The scoreboard is a software component whose role is to determine the functional correctness of the DUT by comparing those responses.

This is the mechanism that checks if each implemented feature by the designers works properly. The scoreboard is actually tightly coupled with the verification plan.
which lists some True/False test conditions allowing to determine if the DUT works (or not).

### 2.1.3 Functional coverage

Next to the scoreboard telling us if everything goes on as expected, we need to have a measure of which features of the design have been tested. It is a very important tool that allow us to decide whether or not the verification of the design is complete. This is very important when using constrained random tests since we cannot know in advance the features that will be tested. This is also the main gauge that allow us to tune the constraints of the tests to orient the randomization to generate stimulus vectors in order to test features that have been less or not exercised.

The verification starts by defining broad constraints for the random tests and by running them. As long as new bugs are found in the design, no change needs to be done regarding the tests, only the RTL needs to be corrected. After going back and forth between the RTL development and the simulations, the number of bugs found will drop, the functional coverage will reach a plateau and won’t increase, even by running the simulation longer. At that point it is time to either change the randomization seeds to generate new stimulus vectors, or analyze the actual coverage to identify holes within the tested features. According to this, new constraints or new tests can be developed to, once more, detect new bugs and finally make the functional coverage evolve.

### 2.1.4 Layered testbench

As already shown in the figure 2.1, a testbench usually wraps around the DUT. It must include all the functionalities required by the tests to perform DUT pin wiggling, data analysis and functional coverage. The more complex is the DUT, the more complex will be the testbench, requiring it to be designed as a complete piece of software, not just a small wrapper around the DUT.

The testbenches need to operate over a wide range of levels of abstraction. At the lowest level it must communicate with the DUT using pin-level interfaces, and at a higher level, it must provide a convenient, easy to use API for the different tests. To be able to operate in that way, most of the testbenches are composed of verification
components (usually, objects of classes) that are themselves categorized in layers. We can distinguish three major layers:

**Transactor layer** This is the level that gathers all the verification components that are directly connected to the DUT. They basically make the conversion between the pin-activity of the DUT and the stream of transactions used by the other verification components to communicate (see section 2.1.5 for an introduction on Transaction Level Modeling). There are three big classes of components fitting in this layer:

- Monitors are monitoring the pin activity and output a stream of related transactions.
- Drivers are converting a stream of transactions and initiate a corresponding pin-activity.
- Responders, just like drivers but respond to a pin activity rather than initiating one.

**Operational layer** This is the layer containing all the components required to generate the stimulus vectors and simulate the DUT environment. Those are components communicating using only transactions and are required to communicate with the transactors if they need to send or receive data from the DUT.

**Analysis layer** It contains all the verification components gathering data to measure the correctness and the completeness of the verification. It contains all the scoreboards and components measuring the functional coverage. Regarding the two others layers, the components within the analysis layer are passive, i.e. they do not produce transactions, they just receive them.

### 2.1.5 Transaction level modeling

This section introduces the fundamental concepts lying behind the Transaction Level Modeling (TLM). Typically, the communication between Verilog modules or VHDL entities is done directly using ports. Each module provides a given number of input and output ports to communicate with others. This approach is very interesting because
2.1 General concepts

Each module can be developed independently from the others. Nevertheless, the simulator needs to simulate every single pin wiggle and event for all the connected ports, which can lead to an increased simulation time if the simulated design involves many modules.

The main idea behind TLM is to replace those low level port events with a few function calls allowing to speedup the simulation. TLM models are communicating with each other using transactions, sending or receiving them through channels. Such a behavior is easier implemented using object oriented languages (SystemC, SystemVerilog, for example) allowing the TLM models to be instantiated by the simulator as object of classes. The transactions also are considered as objects. We can imagine the communication between TLM models as an object (the first TLM model) sending a reference to an object (the transaction) to another object (the second TLM model). Knowing that this transaction can contain a lot of data, this approach is much lighter than using a lot of module ports to transfer that data. This is especially true considering that ports were originally introduced to model the pin interface of real hardware components.

If a TLM model needs to send or receive a transaction, it must make a function call implemented by another TLM model with which it want to communicate. This can create a strong dependency between those two models. TLM implementations thus introduce the concept of TLM ports, allowing to decouple two TLM models just like two modules are decoupled, only relying on their ports to communicate.

When a TLM model wants to make a function call, instead of accessing the function directly (within another TLM model), it make a call to a local function implemented by a TLM port, removing the dependency. A port thus allow a module to make a call of an external function. This imply that others TLM models are implementing those functions and are required to provide an access to them. This access to the functions is done by another kind of TLM ports called exports. Finally internal mechanism of the TLM implementation (like in OVM) allow the programmers to connect a port to an export, routing the function call from the initiator (TLM model making the function call) to the target (TLM model implementing the function) by keeping them to have inter-dependencies.

The figure 2.4 shows the basic connection types available. It always involves two
2.1 General concepts

TLM models, an initiator and a target. Regarding the transaction, one of them is creating the transaction (the producer) while the other is receiving it (the consumer).

Here are some explanations about the different types of connections:

**Put connection** Once the producer has created internally a transaction, it makes a call to the local function `put()` of its put port. This results in a call of the `put()` function in the consumer which, be this mean, will receive the transaction and consume it.

**Get connection** The consumer wants to retrieve a transaction from the producer. To do so, it initiate a call to its local function `get()` of its get port. This will trigger a call of the `get()` in the producer which will generate a new transaction and return it to the consumer.

**Transport connection** This type of connection features both `put()` and `get()` ca-
2.1 General concepts

pabilities at the same time. The initiator creates a transaction and send it as an argument using the `transport()` function or its local transport port. The actual executed function is the `transport()` function of the target which will receive the transaction, generate a new one, and return it as a response. A transport is a bidirectional connection, allowing to send a transaction and to receive another in response.

All the previous situations present an initiator sending or receiving a transaction from a target. This creates a tight synchronization between the two models due to the function itself. A simple way to break this synchronization (if required) is to insert FIFOs that can act as transaction buffers between the producer and the consumer. The figure 2.5 shows such organizations.

![Diagram](image)

Figure 2.5: Different types of TLM connections featuring FIFOs channels

The FIFO are themselves TLM models (and are usually provided by the library implementing the TLM concept like OVM) and are always acting as the target of the function call (i.e. it actually implements the `put()`, `get()` or `transport()` functions and provide them via different exports). Beside the buffer capabilities of such elements, they free the programmer from implementing the `put()`, `get()` and `transport()` functions. He only needs to call them from his TLM model, since the FIFOs are already implementing them.
2.2 OVM key concepts

In this section we will focus on a few concepts implemented by the Open Verification Methodology (OVM) that are used in this project. OVM features much more functionality than those presented here. Please refer to more specialized literature about OVM if you want to get your hands dirty with some code, like the OVM CookBook from Mentor Graphics (Glasser (2009)).

2.2.1 OVM mechanics

2.2.1.1 OVM components

The basic bloc for creating testbenches using OVM is the component. It is constructed from a class, as an object residing in the memory of the simulator. Like any objects, the component needs to be created at run-time, unlike Verilog modules that are created at the elaboration time before the simulation starts.

The architecture of OVM components within a testbench is organized like a tree, having a root item (usually the environment) and as many children as required. The figure 2.6 shows an example of such a hierarchy.

![Figure 2.6: Tree hierarchy of the OVM components](image)

Each OVM component is derived from a base class provided by OVM, `ovm_component`, which features virtual functions and tasks required to develop specialized components. Amongst those, the constructor is called when a new component is created to initialize it and, particularly, to set up the OVM name of the component which is useful to set-up
2.2 OVM key concepts

the component configuration (see 2.2.1.3).

The testbench construction and test execution is divided into phases by OVM. A
phase controller is taking care of calling functions within the OVM components in a
given order to ensure the proper creation of the environment and the execution of the
test. The phases executed by OVM are described here below (given in the order of
their execution):

new  This is not a real OVM phase since it is not controlled by OVM. It however cre-
ates the component in the simulator memory and is thus required before doing
anything else. The first component to be created in a testbench is the root compo-
nent. It is then responsible for creating child components in the hierarchy which
are themselves going to create children. This is a top-down creation mechanism.

build The build phase is the first phase executed by the phase controller, just after
the creation of the component in the memory. It starts by calling the build() function of the component which will gather configuration values and then will
start instantiating child components. Those children, once created, will have
their build() function executed. This building phase will continue until all the
components of the testbench are configured and created. Just like the new phase,
with which it is highly coupled, the building phase is a top-down mechanism.

connect Since the OVM components are using the TLM paradigm, we need to connect
them together. The connect phase is creating the connections amongst all the
components. Unlike the two previous phases, the phase controller is calling the 
connect() functions staring by the leafs of the tree hierarchy. Each connect() function is responsible for connecting the TLM port of the children of the com-
ponent together, as well as connecting it own port to its children. Of course, leaf
components do not need to implement this function since they do not have any
child.

end of elaboration The end_of_elaboration() functions are called once the test-
bench environment is created and connected, ready to be used by a test. Those
functions are just provided as a convenience for the user.
**2.2 OVM key concepts**

**start of simulation** The `start_of_simulation()` functions are called at the simulation time 0.

**run** This is the actual execution of the test. Note, `run()` is a task, not a function. The `run()` task of each component in the environment is forked to be executed in parallel.

**extract** The `extract()` functions are executed in a bottom-up order and are used to extract data from the simulation just after it finishes.

**check** The `checks()` functions are used to analyze and post-process the extracted data.

**report** The `report()` functions generate and output final reports about the execution.

### 2.2.1.2 The OVM factory

The factory is an object oriented design pattern allowing the programmer to have control over the type of created objects during run-time. The factory acts as a substitution to the classic `new()` constructor with a custom `create()` function. This allow the programmer to easily switch the type of some objects without having to modify the whole code that uses them.

In an OVM testbench, the behavior of the components is determined by the procedural code within the phase functions or tasks like `build()`, `connect()` or `run()`. The only degree of liberty we have is to either instantiate or not a component, based on configuration parameters. Nevertheless, one would like to be able to modify the behavior of one component according to the test specificities without having to open the source files and modify the code. The OVM factory serves as a polymorphic constructor allowing the programmer to substitute one component for another by overriding the default type with a new one.

To be able to create objects from a class using the factory, this class must be registered with the factory. Once it is, we can create a new object from the class using the `create()` function which will simply instantiate the constructor of the class. The real advantage of the factory is the ability to override the registered class with another new one. If this override is done, the `create()` function will call the constructor of the
new class instead of the one from the overridden class. In the end the instantiation code remain unchanged but the returned object type is different. The only requirement for the class that is used to override another, is that the former must derive from the latter. This ensure the developed code can remain unchanged, thanks to the polymorphism concept of oriented object programming.

2.2.1.3 Configuration of OVM components

In order to design reusable models, it is important to use a configuration mechanism allowing us to modify and scale the features and capabilities of a component according to the DUT or to the tests requirements. OVM implements the config facility allowing us to pass configuration items to any component within the environment. Each OVM component is associated with a configuration database capable of containing configuration items, which are simply name-value pairs. If we need to configure a component for a test, we simply write a configuration item in its database. That component will be able to retrieve it latter during the build phase while building itself.

In order to be able to fill the database of a target component, we must have a way to uniquely identify it. We already shown with the figure 2.6 that the environment of a testbench is organized as a tree and each component within has an associated OVM name which is given during the instantiation. This name allow us to refer to any OVM component using a path-like string ("env.child1.child2", for example) which is unique by construction. Note that a component may only use such a path to configure its children, it cannot refer to any of its parents.

Setting up configuration items is done using set_config_*() functions, during the build phase of the configuring component. For example, the environment env can configure child2 using

```verbatim
function build();
  ...
  set_config_int("child1.child2", "iterations", 15);
  ...
endfunction;
```

The component child2 will receive in its configuration database the configuration item named "iterations" associated with the integer value 15. During the build
2.2 OVM key concepts

phase, the configuration items can be retrieved using `get_config_*( )` functions. For example, `child2` can recover the previously set configuration item by doing

```systemverilog
...  int iterations; ...
  function build(); ...
      get_config_int("iterations", iterations); ...
  endfunction;
  task run() repeat(iterations)
      $display("Hello world");
  endtask;
```

Here, the example was done using an `int` value but the config facility can be used to pass `string` or `ovm_object` data.

### 2.2.2 Virtual interfaces

In the end, every testbench needs to be connected to the DUT. It means, using OVM, that we must connect SystemVerilog objects, which are dynamic by definition, with a Verilog module which is static. Whereas connecting two RTL modules is easy because their connection, using ports, can be defined statically, we need to find a way around to link the DUT with multiple objects.

One solution is to use a SystemVerilog interface connected to the DUT by sending a virtual interface (i.e. some sort of pointer to the interface) to the OVM components that requires an access to the DUT (usually the transactors).

An OVM testbench has always a top level module in which is instantiated the DUT and the interface just like in the following example.

```systemverilog
// The DUT
module dut (chip_if.dut);
  ...
endmodule

// The interface
interface chip_if();
  logic input_pins [8];
  logic outpu_pins [4];

// Connection for the DUT
modport dut (
2.2 OVM key concepts

In order for an OVM component to be connected with the DUT, we will provide it with a virtual interface pointing to the real chip_if interface. To do so, we simply use the config facility described in 2.2.1.3 by encapsulating the virtual interface in a simple ovm_object and by sending it to the transactors by using the set_config_object() function. Here is an example of such class allowing to encapsulate the virtual interface:

```plaintext
class chip_vif extend ovm_object;
  virtual chip_if vif;
endclass
```

Using this, we are able to send the virtual interface to whom it may concern from the top level module, before the start of the test.

```plaintext
// The top level module, for the OVM testbench
module top ();
  chip_if m_chip_if();
  dut m_dut(m_chip_if.dut);
  chip_vif chip_vif_obj();
  initial begin
    //create the object containing the virtual interface
    chip_vif_obj = new();
    chip_vif_obj.vif = m_chip_if;
    run_test();
  end
endmodule
```
2.2 OVM key concepts

...  
//sending the configuration to the driver (for example)
set_config_object(
    "ovm_top.env.agent.driver",
    "VirtualInterface",
    chip_vif_obj);
...
run_test();
end
endmodule

Using the `get_config_object()` function, the transactors are able to recover the object containing the virtual interface to the real interface, thus allowing them to have access to the DUT.

2.2.3 Tests and stimulus generation

Until now, all the topics we discussed in this chapter were related to the creation or the configuration of testbenches. Next to this, we still need to introduce the components that are using those testbenches. On one side we have the sequences whose purpose is to generate a stream of transactions (random or not), and on the other side the tests, each of them configuring the testbenches and linking one or multiple sequences to it for execution.

2.2.3.1 Sequences

Sequences are the stimulus generators of OVM. They are creating transactions from the transaction classes and generate a stream to be sent to a driver. A sequence is, like any other element in OVM, an object. It is created out of the `ovm_base_sequence` class. The sequences are designed to be executed by a specialized component called a sequencer which is part of the testbench and is usually connected to a driver. Once associated to a sequencer, the method `body()` of a sequence is executed. This method is either generating transactions or may start the execution of other sequences. This latest feature allows us to design a library of low level sequences that can be used as an API to design other higher level sequences.
2.2 OVM key concepts

2.2.3.2 Tests

The tests are actually the components that are called by the `run_test()` task in the top level module of the testbench. They derive from the `ovm_test` class and are intended to configure the testbench and start the execution of selected sequences by some sequencers in the testbench environment. This allows the test to focus on some features of the DUT using the correct configuration and set of sequences.
2.2 OVM key concepts
Chapter 3

SX8723 tests and testbenches

Now that all the required concepts have been introduced and explained, we can jump into the implementation of an OVM testbench targeting the SX8723 chip. Prior to the OVM testbench, the section 3.1 introduce the formerly developed VHDL testbench. We can extract from it basic features and guidelines for the new testbench. There is no use for Semtech to recreate a completely new testbench using novel techniques if their is no advantages. The new approach should at least have the same capabilities as the previous one. As we go through this chapter, the reader will be shown that OVM allows to implement a whole new versatile testbench.

If you are not familiar with the basic concepts of verification, it is recommended to read the chapter 2 prior to this one to have an overview of what can be done with OVM and the concepts used in this project.

3.1 VHDL testbench

The VHDL testbench, even if designed with a language that is not well suited for verification already includes some concepts found in OVM. The figure 3.1 shows the testbench organization. It instantiate two major components: the DUT and a bench driver. The bench driver actually does all the work. It fetches some commands from an external file, interpret them and perform the associated DUT pin wiggling. If a command requires it, some data and information are logged.

We can directly see that there is a clear separation between the testbench (the VHDL component sx8723_tb) and the tests that use the testbench (the *.drv files
containing a series of commands). This is an important concept we explained in the chapter 2 allowing to design tests without having to modify the testbench that wraps around the DUT.

![VHDL testbench diagram](image)

Figure 3.1: Global architecture of the VHDL testbench formerly developed by Semtech.

In other words, the engineers have defined a verification plan specifying a list of features to test and how to test them by making the testbench executing a list of commands contained in the *.drv files. This is the exact definition of directed tests. To have an overview of the whole capabilities of the bench driver, here is a list of the commands it is able to interpret:

- **pause** Performs no operation on the DUT for some time
- **write** Writes a byte of information within a given register of the DUT using the I²C bus and the write format as shown in the figure 1.3.
- **read** Reads a byte of information from a given register of the DUT, using the same I²C bus.
- **force** Writes a digital value on a given DUT pin.
- **check** Reads a digital value from a DUT an compares it to an expected value. The log file reports the result of the comparison.
3.1 VHDL testbench

freq Allows to compare a frequency of a digital signal on a given pin with a given frequency. The log file reports the result of the comparison.

setv Forces a voltage to a given analog pin.

seti Forces a current through a given analog pin.

measurev Compares the voltage of an analog pin with a given value. The log file reports the result of the comparison.

measurei Compares the current flowing through an analog pin with a given value. The log file reports the result of the comparison.

dump Allows to trigger on and off the logging of A/D conversions results. This command tells the testbench to wait for a conversion to be over (using the ready pin as an interrupt signal) and gather both LSB and MSB of the result from the chip registers using the I2C bus automatically.

stamp Writes in the log file the actual simulation time.

Except for the commands freq, measurev and measurei the testbench does not offer the possibility to detect DUT errors during the simulation. The verification team has to post-process the results of the simulations. For example, a test exercise the ADC feature of the chip by providing a sine wave at an analog input. The testbench logs the results of the conversions and they are post-processed using a Matlab program to validate the results using FFT and SNR analysis.

To sum up this overview of the VHDL testbench, we can extract a few good points:

• Separated tests and testbenches.

• Complete API allowing to communicate with the DUT by simple means (a few commands).

However, since the language used is pure VHDL, we can point out a few drawbacks:

• The verification team can only design directed tests. There is no way of designing constrained random tests.
3.2 OVM testbench

- The DUT responses are not checked on-the-fly. The results need to be post-processed (expect for a few, marginal value comparisons).

- The testbench is tied to the DUT. It would require a lot of work to reuse it for another design.

3.2 OVM testbench

This sections presents the whole newly implemented testbench for the SX8723. We will first make a tour of it by describing the global architecture and explaining the implemented functionalities. After that we will go into more details of each implemented components and elements composing the testbench, from the DUT wrappers to the different implemented tests.

The whole implementation was done using the following tools:

- Questasim 6.5a from Mentor Graphics

- Open Verification Methodology, version 2.01

The whole testbench should be portable amongst the other SystemVerilog simulators but, since the DUT is implemented in VHDL, they should be able to handle mixed language simulations. What’s more, the wrappers created to connected the testbench with the real SX8723 model was explicitly designed for Questasim 6.5a (or above) because they need to deal with the mixed language capabilities of the simulator.

3.2.1 Global architecture

The figure 3.2 shows the global architecture of the top level module fullchip_tb implementing the test. Just like we showed in the section 2.2.2 the top level module is composed of:

- the DUT, which is in this case a SystemVerilog wrapper. The is due to the language used to develop the SX8723 (VHDL) which is different from the one used to implement the testbench. The wrapper we can see in lowest part of the figure 3.2 simply provides a mean to connect the DUT to the m_chip_if SystemVerilog interface.
3.2 OVM testbench

- two SystemVerilog interfaces. The first one, called \texttt{m\_chip\_if} gathers all the analog ports from the DUT and provides modports to connect them to the transactors of the testbench. Since the DUT implements also digital ports, this interface include conversion processes allowing the testbench to use the \texttt{logic} type instead of the complex \texttt{analog\_sv} type. The other SystemVerilog interface used is \texttt{m\_i2c\_if} which is using the \texttt{m\_chip\_if} interface to have access to the \texttt{I\textsuperscript{2}C} bus lines (using the \texttt{logic} type of course). This interface implements the behavior of the \texttt{I\textsuperscript{2}C} specification and provides to the transactors an API allowing them to create consistent \texttt{I\textsuperscript{2}C} communications with the DUT.

- the OVM environment defining the complete testbench structure using OVM components.

The OVM environment is a versatile environment, leveraging the OVM config facility, the factory, allowing the different tests to shape it according to their needs. We can distinguish three main parts in the environment:

**Agents** They are featuring the transactors to communicate and analyze the behavior of the DUT. They also include the sequencers allowing to connect them to a stimulus feed (OVM sequence).

**Reference model** This is the golden model, receiving the stimulus transactions and generating expected responses of the DUT.

**Analysis components** They are gathering transactions from the agents and the reference model. They also extract stimulus from the transaction streams coming out of the agents to send them to the reference model, which will latter on induce the generation of transactions from the reference model. The main goal of the analysis components is to perform scoreboarding tasks and functional coverage.

Apart from the reference model, the components are divided in four categories, each of them taking care of some given ports of the DUT.

\footnote{The \texttt{analog\_sv} type will be introduced later in the section 3.2.2, for now, just assume it is the same as the \texttt{analog} type used by the SX8723 VHDL ports.}
Figure 3.2: Global architecture of the new OVM testbench
3.2 OVM testbench

I²C components Those components are in charge of the I²C bus of the chip.

The agent called \texttt{m\_i2c\_agent} includes a monitor which analyze the traffic on the bus and generate a stream of transactions accordingly for the analysis component \texttt{m\_i2c\_analysis}. It also includes a given number of drivers (via the config facility) which may stimulate the bus simultaneously. Each receives a stream of I²C requests (from their associated sequencer or not) and generate an I²C response for each of them\(^1\). The responses are sent back to the stimulus generator and the pairs request/response are sent to the analysis component.

The analysis component, as we said above is simply relaying the requests to the reference model and route the responses from the agent and the reference model to a scoreboard. The requests are also injected in a coverage component providing feedback to the verification engineer.

PIO components Those components are taking care of the power supply of the chip, i.e. they are managing the \texttt{vdd}, \texttt{vss} and \texttt{vpump} pins. Those are purely analog pins, two inputs and one output.

Just like the I²C components we have an agent \texttt{m\_pio\_agent} and an analysis component \texttt{m\_pio\_agent}. The agent includes a driver associated with a sequencer to stimulate the two input pins and send the stimulus transaction to the analysis part. Next to this, a monitor generates a new transaction whenever an event occurs on any of the monitored pins.

As for the analysis component, it perform the exact same operations as its I²C equivalent, but no coverage has been implemented since all the features of the chip are accessed via the I²C bus.

DIG components They have the exact same purpose as the PIO components, but are focused on the general purpose IO pins of the DUT (\texttt{d0} to \texttt{d3}) and the IRQ pin \texttt{ready}. This means the transactors are dealing with mixed signals, thus using analog and digital ports provided by the \texttt{m\_chip\_if} interface.

AC components The AC components are managing analog pins exclusively: the

\(^1\)The I²C requests and I²C response are transactions. They are defined in the section 3.2.4.1
ADC inputs ac2 to ac7. Unlike the two previous set of components, because they are only dealing with input pins, neither a scoreboard nor a monitor is required in the analysis component.

The three last sets of components, managing analog or mixed signal pins, are derived from the same set of classes qualified with the prefix `mixed.*` because their behavior and required functionalities are similar. But we will see that in detail later. Note that, we will refer to those components as the mixed components throughout the rest of the document.

To finish this overview, it might be useful to say now that the reference model is actually a fake one. Due to the limited time allowed for this project, and considering the huge work required to design a complete TLM model of SX8723, the DUT was reused as the reference. To do so, it internally reuse the OVM components of the testbench by configuring them to simply connect the ”reference DUT” with the analysis blocs. This is a great example of the re-usability of the developed components. Another advantage of doing so while creating a testbench, is that if the scoreboard reports an error, this means the testbench is bugged, not the DUT (since the reference and the DUT are both instance of the same VHDL model).

### 3.2.2 DUT wrappers

Before jumping into more details of implementation of the testbench. We should first explain the different mechanisms that are implemented to connect SystemVerilog code and a VHDL DUT. The goal of those wrappers is to ”transform” all the VHDL ports of the DUT, which are using a non standard type `analog` (refer to the section 1.3) to a SystemVerilog interface port allowing to connect the `m_chip_if` interface from the top level module using the `dut` modport. This problem requires us to design two wrappers, just like we can see on the figure 3.3. The first one is a VHDL wrapper, modifying the DUT to make it compatible with the mixed language features of Questasim 6.5a, and the second one gathers all the ports from the VHDL wrapper into a single SystemVerilog port that can be connected to a `chip_if` type interface using the `dut` modport.

The first VHDL wrapper overcome the limitations of the Questasim simulator. As a small reminder, the `analog` type used by the DUT is a VHDL record and is using a
3.2 OVM testbench

Figure 3.3: Internal design of the different wrappers
3.2 OVM testbench

resolve function. The package that goes with it also features a bunch of procedures to manipulate it. Reusing this package directly within a SystemVerilog code was impossible since the used version of Questasim does not allow to use a VHDL type that has a resolve function associated directly from a SystemVerilog code. What’s more, all the procedures from the `analog` package become completely unusable from SystemVerilog. To bypass those two problems, we had to do two things:

- First, we made a copy of the `analog` type called `analog_sv` in a new VHDL package with the small difference that the new type does not have a resolve function and thus is suitable to be used at the boundary of the two languages.

The following piece of code shows the new VHDL package. As you can see, it is very short. It can be seen as an extension to the previously designed package from Semtech.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library xemics; -- Semtech's library
use xemics.analog.all;
-- real32, model_type and device_type are
-- defined in the xemics.analog package.

package analog_wrapper is
  type analog_sv is record
    v_value : real32;
    i_value : real32;
    z_value : real32;
    model : model_type;
    device : device_type;
    nb_gene : natural;
    nb_load : natural;
  end record;
end package;
```

The VHDL wrapper is simply using this new type for its own ports, allowing it to be instantiated from a SystemVerilog module. Internally, the VHDL wrapper is instantiating the real DUT. For each port of the DUT, the wrappers creates two `analog_sv` ports just like shown in the figure 3.3. One `input` port to write a new value from the outside, and one `output` port allowing to read the value. This split of each port is required because we lost the resolve function. Using `inout` ports would require that function because multiple drivers could operate
the port at the same time. Splitting each port into two distinct ports allow us to use the wrapper from SystemVerilog but it limits the access to the ports to a single driver (using the *input* ports) and multiple readers (using the *output* ports).

- Secondly, we had to recreate the SystemVerilog equivalent of the procedures manipulating the original *analog* type. A SystemVerilog package was created to re-implement the VHDL procedures and functions by the mean of SystemVerilog tasks and functions.

```verbatim
package analog_pkg;
    import analog::*;
    import analog_wrapper::*;

    function automatic real check_z (real z);
        ...
    function automatic bit is_high_z (analog_sv a);
        ...
    task automatic v_gene ( inout analog_sv a,
                              input real32 v_value, z_value);
        ...
endpackage
```

As you can see, the re-implemented functions and tasks are using the new *analog_wrapper* package, because they are manipulating *analog_sv* types now. It also reuse the original *analog* package giving access to some constants defined inside.

As for the SystemVerilog wrapper, it simply instantiate the VHDL wrapper and gathers all the *analog_sv* ports into a SystemVerilog interface port. This allow us, from the top level module of the testbench, to connect the internal variables provided by the *dut* modport of a *chip_if* interface to the ports of the SystemVerilog wrapper without having to connect them one by one (please review the basics of SystemVerilog interfaces if this in unclear).
3.2.3 SystemVerilog interfaces

The SystemVerilog interfaces are the links between the OVM transactors and the DUT. As we will see, they are not just a bunch of signals connecting them, but includes some mechanisms for the transactors to operate those signals easily and in a coherent way.

3.2.3.1 Chip interface

The chip if interface is the lowest level interface in our system. It is the unique access point to the DUT ports. We already know that many of them are mixed signal ports like d0 and d1, or event purely digital like the remaining ports of the GPIO, the ready port and of course the I²C bus lines. The ease the job of the transactors, the chip interface is implementing conversion mechanisms for those ports, allowing the OVM transactors to manipulate logic types when the port is digital. The figure 3.4 shows the conceptual implementation of the chip interface.

The chip interface is featuring the following modports:

- The dut modport, allowing the connection with the DUT.

- pio_driver, pio_monitor, dig_driver, dig_monitor, ac_driver and ac_monitor modports. Those modports are to be used by the corresponding transactors in the mixed components.

The chip interface also have a port (just like a typical module port) that can be used to connect an I²C interface using its extern modport (see section 3.5).

We previously explained that each port of the SX8723 was divided into two ports accessible from the wrapper. If a transactor wants to write a value to a port, it needs to use the input port. Reversely, if it wants to read a value, it must be done using the output port. For example, to set up the power supply of the chip, the driver of the AC agent must write an analog_sv value to the vdd_in port of the DUT. The value is transfered to the DUT via all the wrappers where the real value of the VDD pad is resolved. This computed value makes all his way back to the vdd_out port. This is a simple case where only analog values are involved. The role of the chip interface is simply to link the ports of the DUT wrapper (available via the dut modport) to any of the mixed modport.
3.2 OVM testbench

Figure 3.4: Internals of the chip_if SystemVerilog interface
However, even if all the ports of the DUT wrapper are of type `analog_sv`, some of them might carry digital values. To ease the transactor operations, the chip interface implements converters allowing them to directly write or read `logic` type values. Those converters are connected in parallel to the classic analog connections, just like shown of the right side of the figure 3.4. Finally, the transactors that are managing mixed value ports are able to read and write `logic` and/or `analog_sv` values according to their needs.

Until now, we left the I²C bus ports apart because the way they are managed inside the chip interface is a bit different. The chip interface still inserts the `analog_sv` to `logic` (and vice-versa) converters, but the digital access to the bus is not connected directly to any I²C transactors. Instead, it is connected to another SystemVerilog interface implementing the I²C bus behavior.

However only putting the converters will lead to a incorrect behavior of the I²C bus. For example, if the sda line is forced to 0, the I²C interface will write a 0 value in the input sda port of the DUT wrapper (via the D2A converter). This value will be processed at the level of the RTL model and will directly be written to the output sda port. Finally, the A2D convert will also output a 0. But we don't know if this 0 is a value written by the DUT itself or by other components on the I²C bus. So we need to insert another converter after the A2D that will output what is really written by the DUT. If this converter receives a 0 like in our example, it must check if it is a 0 forced by the DUT or not, generating a 0 or a 1 at its output respectively. The same mechanism must be applied for the scl line.

To illustrate all those explanation about the chip interface, here is a small piece of code taken from its implementation:

```verilog
interface chip_if (i2c_if.extern_mp m_i2c_if);
    // Analog_sv variables
    // *_in are used to write an analog_sv value to the DUT
    // *_out are used to read an analog_sv value from the DUT
    analog_sv ac_in [7:2], ac_out[7:2];
    analog_sv d_in [3:0], d_out [3:0];
    analog_sv ready_in , ready_out;
    analog_sv scl_in , scl_out;
    analog_sv sda_in , sda_out;
    analog_sv vdd_in , vdd_out;
    analog_sv vpump_in , vpump_out;
    analog_sv vss_in , vss_out;
```

40
// logic (digital) variables
// _in_dig are used to write a logic value to the DUT
// _out_dig are used to read a logic value from the DUT
logic sda_out_dig = 1, sda_in_dig;
logic scl_out_dig = 1, scl_in_dig;
logic d_in_dig [3:0], d_out_dig[3:0];
logic ready_in_dig , ready_out_dig;

// Modports
modport dut (    
    ref ac_in, ac_out, d_in, d_out, ready_in, ready_out,
    scl_in, scl_out, sda_in, sda_out, vpump_in, vpump_out,
    vdd_in, vdd_out, vss_in, vss_out
);

modport pio_driver (input vss_out, vdd_out, vpump_out,
    output vss_in, vdd_in)
);
modport pio_monitor (input vss_out, vdd_out, vpump_out);
modport ac_driver (input ac_out, d_out,
    output ac_in, d_in)
);
modport ac_monitor (input ac_out, d_out)
);
modport dig_driver (output d_in_dig, d_in, ready_in_dig,
    input d_out_dig, d_out, ready_out_dig)
);
modport dig_monitor (input d_out_dig, d_out, ready_out_dig)
);

... 
// The converters between the analog_sv variables
// and the logic variable go here
...
endinterface

3.2.3.2 I2C interface

The goal of the I2C interface is to implement and model the behavior of an I2C bus based on the official specification. It also provides a complete set of tasks to operate it. The figure 3.5 shows the internal architecture of the interface. Note that the shown schematic is applied twice, once for each of the two bus lines: sda and scl.

Internally, the interface has 2 wires named sda and scl that, obviously, represents the I2C bus lines. Each wires has also a table of logic variables associated that represents all the connected line drivers (basically, one for each component connected on the bus). All those drivers value are ANDed to model the behavior of the I2C bus lines. When any line driver applies a 0, the line should be forced to 0, while when it applies a 1, the line is released. Note the line drivers are not to be confused with the...
3.2 OVM testbench

virtual connections to the OVM I2C transactors

Figure 3.5: Internals of the i2c_if SystemVerilog interface
OVM drivers. Each transactor and the DUT itself may have its own line driver within the I²C interface, allowing it to drive the bus.

This interface provides three modports:

- An `extern_mp` modport to be connected to the chip interface port. It gives access to the digital input and output port of the sda and scl pins of the DUT provided by the chip interface, namely `sda_inDig`, `sda_outDig`, `scl_inDig` and `scl_outDig`.

- A `master_mp` and `monitor_mp` modport to be used by the drivers and the monitor of the I²C agent. They only give a read access to the bus state and events.

One surprising thing on the figure 3.5 is that no signal is provided to the I²C transactors via the modports. This force the transactors (and thus the programmer) to only use the set of tasks provided by the I²C interface to write and read the bus. All the tasks have been designed to operate together, creating sda and scl waveforms respecting the I²C specification. Allowing the transactors to have direct access to the sda and scl line might disturb this balance and the modeled bus by the I²C interface might simply not work.

Regarding those tasks, we have a task to register the OVM drivers within the interface (i.e. to create the associated sda and scl line driver, and structure to handle its scl clock). There is also a set of tasks for I²C masters allowing them to send start bits, stop bits and read or write bits on sda line while generating the scl clock. Another set of tasks for I²C slaves allow them to simply read bits from the sda line. Note that the I²C interface provide two versions of the read tasks: one is for reading the real sda line, while the other is for reading the `sda_outDig` port from the chip interface (i.e. reading only the DUT response).

### 3.2.4 Transactions

In this section we will introduce the different types of transactions that are used by the OVM components to communicate.

---

1. The I²C specification says that each I²C master has to generate its own clock while communicating. The I²C interface automatically generate the clock when a master is communicating. I.e. between a start and a stop bit.
With OVM, all the classes used to create transactions should derive from the base class `ovm_transaction`, while the sequences should create transactions from classes that derive from `ovm_sequence_item`. In order to avoid creating multiple classes that do the same thing, all the classes developed to create transactions for this testbench are derived from the `ovm_sequence_item` class. This can be done since that class itself is derived from the `ovm_transaction` class, considering the concept of object polymorphism. Internally, a class to create transactions is always constructed using the same scheme. First we declare some object variables. They will contain the actual information that carry the transaction. Along with them a number of methods are defined to manipulate those variables. The most important methods, and the one we used in our case are:

- `copy()`, used to copy another transaction (of the same type).
- `clone()`, used to create a clone of the transaction.
- `convert2string()` used to generate a string that can be displayed in the console or a log file.
- `comp()` used to compare the transaction with another of the same type. This is mainly used by the scoreboard to compare transactions issued from the DUT and the reference model. It can be interesting to take a peek in the code to see the implementation of that function for the transactions used in our testbench.

All those could be implemented automatically by using OVM macros, but we choose to implement them ourselves because it allows us to have full control over their behavior. Regarding the OVM macros, we explicitly try to avoid using them, not only for implementing the transaction classes but for the whole OVM part of the project.

### 3.2.4.1 I2C transactions

In order to define transactions for the I²C bus, we had to consider the fact that we want to be able to handle erroneous waveforms on the bus lines to a certain extent. This allow the testbench to be used to test the DUT for bus and protocol error management. Ultimately, the transactions need to be converted into pin wiggles and vice-versa by the transactors. This is done by using the API provided by the I²C interface designed
to operate the bus lines, thus defining the lowest operations accessible. All this means, the information contained in an \( I^2C \) transaction should not represent too high level informations, preventing us from injecting wrong crafted data streams on the bus. But it should not be too low level either, preventing, in that case, the transaction to be translated into task executions on the \( I^2C \) interface.

We decided to define two sets of transactions: the \( I^2C \) requests and their associated \( I^2C \) responses. The \( I^2C \) requests, created from the `i2c_req_item` are transactions mainly used by the different stimulus generators (the sequencers for example) to initiate an operation on the bus. The following code snippet shows the declaration of the object variables within this class:

```verilog
class i2c_req_item extends ovm_sequence_item;

  // object variables
  purpose_req purpose;
  logic [7:0] data = 'x;
  logic ack = 'x;
  integer max = 8;

endclass
```

The `purpose` variable defines the meaning of the request. It is based on the `purpose_req` enumerated type defining the following purposes:

**START** It is used to represent a start or a repeated start bit as defined in the \( I^2C \) specification. The other object variables are not used in this case.

**STOP** Same as above but for the \( I^2C \) stop bit, indicating the end of an \( I^2C \) communication between a master and a slave.

**SEND_ACK** It is used to represent the action of sending an acknowledgement, usually after having received a byte of information from the bus. The value of the acknowledgement is contained in the `ack` variable.

**SEND_DATA** It is used to represent the action of sending multiple bits on the sda line, up to a complete byte (8 bits). The data is contained in the `data` variable and the `max` variable contains the size of the data in bits. If `max` is lower than 8, only the most significant bits of the vector `data` are considered. This is due to
the fact that, when sending a byte on the sda line, the most significant bit is sent first.

**READ_DATA** It represents the action of reading a given number of bits from the sda line. This number is specified in the max variable, which can be set to a maximum of 8.

**READ_ACK** It represents the action of reading an acknowledgement on the bus. (It can be seen as the degenerate form of the previous request purpose when the amount of bit to read is set to 1).

Those requests may either be generated by the monitor for the analysis components, or by a stimulus generator for the reference model and the driver. In the latter case, the drivers generate an I²C response for each request to be sent back to the stimulus generator. The following portion of code shows the object variable from the i2c_rsp_item used to generated those responses:

```plaintext
class i2c_rsp_item extends ovm_sequence_item;
  ...
  purpose_rsp purpose;
  // object variables
  logic [7:0] data = 'x;
  logic ack = 'x;
  integer max = 8;
  ...
endclass
```

This is somehow very similar to the request class, even if the intended purposes are different. The purpose variable might have one of the following values:

**DONE** It is used to acknowledge the stimulus generator when it send a START or a STOP request.

**MATCH** It is used as a response to the SEND_DATA and SEND_ACK requests to indicate that all the bits has been correctly been sent.

**MISMATCH** Unlike the previous one, it is used to indicate that the data has not been correctly sent. The source of this type of transaction comes from the arbitration

---

1. The request and response classes are very similar. An improvement that can be done is to make those two classes derive from a base class gathering the common variables.
rules defined by the I²C specification when two or more masters are operating on the bus at the same time. If such a response is received by a stimulus generator, it means it has lost the arbitration process and thus, the right to communicate with the slave.

**ACK** It used to represent the response to a READ.ACK request. The value of the acknowledgement is contained on the ack variable. However, it is only valid if the variable max is set to 1. Otherwise it means the request was processed by the driver when a device was initiating a repeated start or stop bit on the bus, which invalidate the read value.

**DATA** It is used to represent the response to a READ.DATA request. The read bits are contained in the data vector, while the max variable indicate how many bits have been read successfully and is at most equal to the value of the max variable of the request. If it is lower, it means a repeated start or a stop bit occurred on the bus preventing the request to be completed. Again the max variable indicate the number of valid bit starting for the most significant one.

The possibility to send or read a given number of bits from the I²C was introduced to allow stimulus generators to create an erroneous bit stream on the sda line.

In order to use the I²C bus, a stimulus generator has to send a series of requests, guided by their responses. Any kind of series may be generated: if we follow the I²C specification, we can create a correct I²C communication, or we can manipulate the drivers to drive the bus wrongly, allowing to check the behavior of the DUT in such cases.

The last designed transaction for the I²C part of the testbench is a simple container, linking a request and its response in a single transaction. Those are created from the i2c.pair.item class. They are mainly used by the I²C monitor, when it identify a request and a response on the bus, it creates a pair transaction and send it to the analysis components.
3.2 OVM testbench

3.2.4.2 Mixed signal transactions

The mixed signal transactions need to be able to handle either analog or digital values. Unlike the I²C transactions, they do not take part in any kind of protocol, but need to represent the value of a port at a given time. They are used by mixed components (PIO, AC and DIG components). The drivers are receiving a mixed request, created by a stimulus generator from the mixed_req_item class, and send back a response if required (created from the mixed_rsp_item class).

Those two classes are derived for the same base class called mixed_base_item. That class is also used as is by the different monitors since the transactions they are creating are neither a request nor a response. Here is part of the code showing the object variables:

```cpp
class mixed_base_item extends ovm_sequence_item;
...
  base_e base = UNKNOWN;
  string id = "";
  shortreal val_ana = 0.0;
  bit highz = 0;
  logic val_dig = 0;
  real stamp;
...
endclass
```

While the id string defines what pin of the DUT is associated with the transaction, the base variable define the type of information carried. It may have one of the following values:

**ANALOG** In this case, the transaction represents an analog value. The information is put in the val_ana and highz variables. The first one contains the analog value of a pin and the second one determine if the represented value has to be associated with a high impedance pin (in which case the former variable in not used)

**DIGITAL** In this case the transaction represents a digital value carried by the val_dig variable.

**UNKNOWN** This case is only used when a transaction is created by mixed monitors.

This come from the fact that a monitor does not know in advance if it should
3.2 OVM testbench

read a digital or an analog value out of a mixed signal port of the DUT, thus reading both.

The stamp variable may contain a simulation time. It is only set and used when a transaction is created by monitors to indicate the simulation time at which occurred the event that led to the transaction creation.

As for the requests, created by stimulus generators (the sequences, for example), we already said they are derived from the base class, as shown in the following code:

```cpp
class mixed_req_item extends mixed_base_item;
  ...
  purpose_e purpose = READ;
  ...
endclass
```

A new variable named purpose appeared. Just like in the I²C requests, it defines the operation that the drivers should do when they receive a request. We can have the following request purposes:

**SEND** Indicate that we want to write a value to a DUT port. The base variable determine if we want to write an analog or a digital value, and the id is used to select the targeted DUT port.

**READ** Indicate that we want to read a value from a DUT port. Just like previously, the targeted port is set using id and the type using base. Such a request will induce the driver to generate a response from the mixed_rsp_item class containing the requested data.

**READ_WAIT** Just like READ, but the driver will not create a response immediately. It will wait for an event (value change) on the targeted DUT port to generate the response.

The class used to generate responses is implemented as follow:

```cpp
typedef mixed_base_item mixed_rsp_item;
```

This is actually a simple alias of the mixed_base_item since responses do not need extra variables nor to redefine any functions. However, the drivers will never generate UNKNOWN responses, since the response type is defined by the request, which may not request a read of type such a type.
3.2 OVM testbench

3.2.5 OVM components

We already showed in the global testbench architecture that we have two types of high level OVM components: the agents and the analysis components. The following sections will describe those components in more details than done before.

3.2.5.1 I2C agent

The I²C agent is a highly versatile OVM component. It has been designed with reusabilty in mind and is absolutely not specific to the DUT. The only requirement of this agent is that its transactors must be passed a virtual interface pointing to a i2c_if interface, which is chip independent too.

The I²C agent features up to five types of OVM components (its children) that are detailed below. The existence of those components within the agent and their amount is specified when the OVM environment is configured using the OVM config facility. The figure 3.6 show the default configuration of the environment for this agent.

![Diagram of I²C agent](image)

Figure 3.6: I²C agent as implemented in the default environment
I2C master  This component is one of the two transactor types of the I2C agent. It features a transport export to receive I2C requests and send back the associated responses. The following piece of code shows the skeleton of the `run()` task.

```verilog
task run();
   // Register the i2c_master with the i2c_if SV interface.
   id = this.get_full_name();
   m_i2c_if.register(id);

   // Keep fetching new transactions (requests)
   forever begin
      ...
      transport_channel.get_request_export.get(tx);

      // operate the bus using m_i2c_if tasks
      // generate the i2c response
      case (tx.purpose)
      START:
      ...
      STOP:
      ...
      SEND_ACK:
      ...
      READ_ACK:
      ...
      SEND_DATA:
      ...
      READ_DATA:
      ...
      endcase

      // Send back the response
      transport_channel.put_response_export.put(rx);
   end
endtask
```

This code actually fetches an I2C request from an internal channel `transport_channel` (the end point of the transport connection, which implement the `put()` and `get()` functions), analyze its purpose and drive the bus accordingly using the API provided by the I2C interface. In the mean time, an I2C response is generated and send back through the transport export via the same internal channel. We used a transport export because each request requires a response as defined by the I2C transactions we introduced earlier, otherwise we would have decoupled the stream of requests from the stream of responses by using a distinct get and put export.

Within the I2C agent, the `i2c_master` component might be instantiated as many...
time as the user wants by using the configuration mechanism of OVM. Each transport export of each `i2c_master` is routed to the boundary of the agent, read to be used by external components, like a TLM model of a I²C master component (microcontroller, ...)

One could wonder why this transactor is not called `i2c_driver` because it exactly does the operation of what we referred earlier as a driver. We chose to keep that name for the components that are connected to an OVM sequencer. The `i2c_master` name comes from the fact that it mimics the low level operation of an I²C master component as indicated in the I²C standard.

**I2C monitor** This component can be instantiated only once in the agent. Its primary goal is to monitor the I²C bus line using the I²C interface tasks. It is assuming that the I²C communication on the sda line is correctly formed. In other words, a start bit is followed by an undetermined series of 9 bits (a byte of data plus a bit for the acknowledgement). The only bus error that this monitor is able to detect is if the stop (or repeated start) occurs in the middle of a set of 9 bits, it is able to generate a transaction containing missing data as provided by the I²C transaction definitions.

The I²C monitor is always sensing for a start (or repeated start) bit which indicate a new communication (even if the previous one is not over). After that, by analyzing the bit stream on the sda line, it is able to detect if the I²C master is targeting a I²C slave for reading or writing. Out of those information, it is able to generate multiple a pairs of I²C transactions (composed of the request from the master and the slave response) that it encapsulate in `i2c_pair_item` transactions. Those pairs are sent through an analysis port, that is routed by the agent to its own boundary for analysis purposes.

It might seem obvious, but this I²C monitor is only able to apprehend the I²C communications from all the components connected on the bus. It cannot only focus on the DUT.

**I2C driver** This component is a wrapper for the `i2c_master` class described here above. It is used to convert the transport export into a particular port of OVM that allow to connect a sequencer (that will execute sequences to generate stimulus). That particular port of type `ovm_seq_item_pull_port` comes from the `ovm_driver` class from
which `i2c_driver` is deriving.

It also features two analysis port:

- **`req_ap`** is used to send the request as soon as it has been fetched from the sequencer. This ensure that the request is sent to the analysis component within the same simulation time step.

- **`pair_ap`** is used to send a `i2c_pair_item` after a response has been received from the internal instantiation of `i2c_master` to the request.

From the agent point of view, the number of I²C drivers instantiated can be tuned using the OVM config facility. If no internal sequencer of the agent is associated with a driver, the agent route the `ovm_seq_item_pull_port` port to its boundary. This allow the external environment to connect its own sequencer if required.

### I²C sequencer

The I²C sequencers are created from the `i2c_sequencer` class. It is deriving from the special OVM class `ovm_sequencer` and is specialized to execute a sequence that is generating `i2c_requests` transactions and waiting for `i2c_response` ones. The execution of the sequences is completely handled by the base OVM class and the programmer has nothing to do, except associating a sequence with a sequencer from the test classes.

### I²C talker

The talkers are very simple components. They are deriving from the `ovm_subscriber` class. This particular OVM class executes a `write()` function each time a transaction is received from the input analysis export that can be connected to any analysis port as long as they are specialized for the same type of transactions. In our case, the talkers are receiving a stream of `i2c_pair_item` transactions and are displaying them either into the transcript or a log file (the name of the log file can also be specified with the OVM config facility).

### 3.2.5.2 I²C analysis component

This I²C analysis component is designed to gather streams of transactions that are coming out of the agent and to communicate with the reference model. The figure 3.7
show the internal connections between components composing it. We can see that we have two input exports on the bottom of the picture. One is receiving \texttt{i2c.pair.item} transactions from the \texttt{I2C} monitor or any of the \texttt{I2C} driver. Those transactions are going straight to the scoreboard for comparison with reference transactions. The other is receiving \texttt{I2C} requests from one of the drivers in the agent. Those requests are going to the adapter which will transmit them to the reference model and also to the coverage component which gathers information about the functional coverage. We can also see that the analysis component has two exports (on the top of the figure) to communicate with the reference model.

![Diagram of I2C analysis component](image)

\textbf{Figure 3.7: I2C analysis component as implemented in the default environment}

Let’s now focus more on the internals of each component within the analysis component. Note that the scoreboard and the coverage components can be removed (i.e. not instantiated) using OVM configuration items.
3.2 OVM testbench

I2C adapter The adapter is simply a routing device for transactions. It is receiving I2C requests from a driver. Those requests are, at their entrance in the adapter, buffered by a FIFO (which act as the ending point of the TLM connection). However, a process is constantly monitoring that FIFO and each time a request is available, the request is removed from it, and put into another internal FIFO available for reading by the reference model via a get export.

The reference model in responsible for monitoring that FIFO via the export and should get the requests as soon as they are available. Considering all this, and the fact that I2C drivers send requests through their analysis port in no simulation time step, the reference model is able to receive every one of them at the exact same simulation time as the I2C drivers. So, the DUT and the reference model are stimulated simultaneously. Once that the reference model has processed a request it send back a response using the input export of the adapter an put it into an internal FIFO of the adapter.

Finally, the adapter has a request and its associated response. It is able to send a i2c_pair_item transaction containing both of them to the scoreboard via it analysis port.

The following code shows the a part of the run() task of the adapter and might clarify what has just been explained:

```verilog
fork
  forever begin
    // fetch a request if available
    analysis_fifo.get(_req);
    // clone it
    $cast(req,_req.clone());
    //send it to the reference model
    // (put_port is an access point to an internal channel)
    put_port.put(_req);
    //send the clone in another fifo for internal use
    fifo.put(req);
  end
  forever begin
    // get a response from the reference model
    get_port.get(rsp_ref);
    // create a pair item
    pair = new(null, rsp_ref);
    // get the clone of the request from the internal fifo
    fifo.get(pair.req);
    // send the pair via the analysis port
    analysis_ref_port.write(pair);
  end
end
```
The FIFOs for communicating with the reference model could have been instantiated either in the reference model or inside the adapter. We choose the latter solution because it avoids the designer of the reference model to handle this. Putting them in the analysis component allow him to only make function calls (get() to receive a request and put() to send back a response) without been required to implement them. It also frees him from effectively making the FIFOs instantiations because it is automatically done by the mixed analysis components.

**I2C scoreboard** The scoreboard is receiving i2c_pair_item transactions from the agent and from the reference model using two input exports connected internally to two FIFO. The actual implementation of the scoreboard is fetching a pair transaction from the agent, another from the reference and compare only the I^2^C responses inside them. This comparison is done using the comp() function that is implemented in the class used to created I^2^C response (see 3.2.4.1).

The scoreboard is instantiated within the analysis component using the OVM factory:

```cpp
if (has_scoreboard)
    scoreboard = i2c_scoreboard::type_id::create("scoreboard", this);
```

This allows the test designer to swap this simple scoreboard to another that is more adapter to a given test.

**I2C coverage** The coverage component is simply gathering I^2^C requests from its input. It is designed like the I^2^C talker, using the ovm_subscriber. This means each time a request is received, the function write() is executed. The requests are sampled by covergroups that gathers coverage data about what is happening on the I^2^C bus.

For now, only two covergroups have been designed:

- The first one, called purp_cov, gathers information about the purpose of the requests and checks that each of them have been seen at least once. If so, a coverage of 100% is achieved.
3.2 OVM testbench

- The second covergroup, called `req_cov` is a bit more involved with the chip itself. It actually monitors which registers of the SX8723 have been accessed and if it was for reading or writing. A coverage of 100% can be reached if each register has been accessed twice, one for reading and one for writing, at least.

Just like the scoreboard, it is instantiated using the factory. So it can be replaced easily according to the tests without modifying the code.

3.2.5.3 Mixed signal agent

Just like the I²C agent, mixed signal agents are high level components gathering transactors and some talkers. They somehow are simpler because there is no possibility to instantiate multiple drivers inside. However, the `mixed_agent` is not designed to be instantiated as is. It requires to be specialized by specifying which type of monitor, driver and talker should be used. To illustrate this, here is the class declaration:

```plaintext
class mixed_agent #(type monitor_type = mixed_monitor,
                   type driver_type  = mixed_driver,
                   type talker_type  = mixed_talker)
  extends ovm_agent;
...
```

And the class definition for the PIO agent (we have the equivalent for the DIG agent and AC agent):

```plaintext
class pio_agent extends mixed_agent #(pio_monitor, pio_driver, mixed_talker);

  function new (string name, ovm_component parent);
    super.new(name, parent, "pio_vif");
  endfunction
endclass
```

As those few lines suggest, the OVM environment is instantiating the PIO agent from the `pio_agent` class. They also show us that the classes used to instantiate the monitor and the driver are also deriving from base classes. In our example, the driver is created from the `pio_driver` which itself deriving from the `mixed_driver` class, so is the `pio_monitor` from the `mixed_monitor`.

The figure 3.8 shows the default configuration for each agent in the environment (note that the AC agent does not need the monitor nor it’s associated talker).
We are now going to focus on the base classes used to create internal components of the agents.

**Mixed driver**  As a base class, its `run()` task is empty and virtual. It need to be re-implemented by the specialized class according to the pins it will operate on and to the type of those pins (input, output or both, analog, digital or mixed signal). To ease the creation of those specialized classes, the `mixed_driver` class is already implementing the TLM ports, fetching the configuration items and providing a list of tasks to operate the DUT port chip interface and the `driver` modport.

The communication with the stimulus generator can be done using the `seq_item_pull_port` if a sequencer is used. The response, if any, is sent back using the same port. The driver can also be configured to not use this port but to use a couple of exports instead: an `ovm_put_export` and an `ovm_get_export`. They can be sued to to connect another type of stimulus generator than the OVM sequencers.

As for the implemented tasks, a complete set has been developed to read or write data on a given port of the DUT according to its type (analog, digital or mixed signal).
Let’s take a peek at a couple of those tasks. The first one is used to read a mixed signal port. Remember that for such ports, the chip interface if providing to the driver an access for reading the digital value and another for the analog value. The following task is taking those two access as arguments (\(b\) and \(a\)). The reference to the the mixed request is also passed (but it has already been detected to be a \texttt{READ} or \texttt{READ\_WAIT} request) and a reference to an empty mixed response is also required. According to the type of reading (\texttt{ANALOG} or \texttt{DIGITAL}), a response transaction is generated by reading the access \(a\) or \(b\). Here is the code to help with the previous explanation:

```verilog
task automatic process_read (input logic b, ref analog sv a,
mixed_req_item _req, mixed_rsp_item _rsp);
    case (_req.base)
        DIGITAL: mixed_base_item::gen_tr_dig(_rsp, b, _req.id);
        ANALOG: mixed_base_item::gen_tr_ana(_rsp, a, _req.id);
        default
            ovm_report_warning ("driver",
                                "Unknown base (should be DIGITAL or ANALOG)");
    endcase
endtask
```

The second example is a task used to write an analog value on a port only. The reference to the input port of the DUT from the chip interface is passed as an argument as well as the request (already known to have a \texttt{SEND} purpose). Once the task is executed, it drives the analog value contained in the request to the analog input port of the DUT, as such:

```verilog
task automatic process_send_ana (ref analog sv a, mixed_req_item req);
mixed_base_item _req = req;
    case (_req.base)
        DIGITAL: ovm_report_warning("driver", "Cannot drive digital pins");
        ANALOG: mixed_base_item::drive_ana(a, _req);
        default
            ovm_report_warning ("driver",
                                "Unknown base (should be ANALOG)");
    endcase
endtask
```

To illustrate a bit more the concept of specialization of the \texttt{mixed\_driver} class, let’s take the example of the PIO agent which is using those tasks:

```verilog
task run ();
```
forever begin
  ...
  // Fetch the request
  if (no_seq_port)
    channel.get_request_export.get(req);
  else
    seq_item_port.get(req);
  // send a copy of the request to the analysis port
  $cast(_req,req);
  analysis_port.write(_req);
  case (req.purpose)
    SEND:
      begin
        case (req.id)
          "vdd": process_send_ana(m_chip_if.pio_driver.vdd_in, req);
          ...
        endcase
      end
    READ, READ_WAIT:
      begin
        case (req.id)
          ...
          "vpump":
            begin
              if (req.purpose == READ_WAIT)
                @(m_chip_if.pio_driver.vpump_out.v_value,
                m_chip_if.pio_driver.vpump_out.device);
                process_read_ana(m_chip_if.pio_driver.vpump_out, req, rsp);
            end
          ...
        endcase
        //send the response
        if (no_seq_port)
          channel.put_response_export.put(rsp);
        else begin
          rsp.set_id_info(req);
          seq_item_port.put(rsp);
        end
      end
  ...
  endcase
end
endtask

To finish this explanation about the drivers, they are sending a copy of the request...
to an analysis port right after it has been fetched, just like the \( \text{I}^2\text{C} \) drivers. So the analysis components are aware of the request in no simulation time step.

**Mixed monitor**  The monitor is based on the exact same principle as driver. The TLM port is defined (in this case a simple analysis port), a set of tasks is implemented to read a given DUT output port and generating a transaction. The \texttt{run()}\ task is empty and should be re-implemented be specialized classes too.

Here is one of the implemented tasks that must be used in the re-implementation of the \texttt{run()}\ task:

```plaintext
task automatic process_read (inout logic b, ref analog_sv a, input string id);
    mixed_base_item tr = new();
    #0 mixed_base_item::gen_tr(tr, a, b, id);
    analysis_port.write(tr);
endtask
```

Remember that the generated transaction is derived from the \texttt{mixed_base_item} and is of type \texttt{UNKNOWN} in that case. There are also two other tasks \texttt{process_read_ana()}\ and \texttt{process_read_dig()}\ that generate a transaction carrying an \texttt{ANALOG} or \texttt{DIGITAL} type data respectively. Those tasks take also the responsibility to send the generated task directly through the analysis port.

Here is an example of specialized monitor (PIO monitor here) that re-implement the \texttt{run()}\ task of the \texttt{mixed_monitor} class:

```plaintext
task run ();
    fork
        forever @(m_chip_if.pio_monitor.vdd_out.v_value,
                 m_chip_if.pio_monitor.vdd_out.device)
            process_read_ana(m_chip_if.pio_monitor.vdd_out, "vdd");
        forever @(m_chip_if.pio_monitor.vss_out.v_value,
                 m_chip_if.pio_monitor.vss_out.device)
            process_read_ana(m_chip_if.pio_monitor.vss_out, "vss");
        forever @(m_chip_if.pio_monitor.vpump_out.v_value,
                 m_chip_if.pio_monitor.vpump_out.device)
            process_read_ana(m_chip_if.pio_monitor.vpump_out, "vpump");
    join
endtask
```

**Mixed talker**  Unlike the two previous detailed classes, the \texttt{mixed_talker} class does not obviously need to be specialized. But the mixed agent is ready to use a specialized version.
Just like the I^2C talkers, the mixed talkers are receiving a stream \texttt{mixed_base_item} transactions and output them in the transcript or a log file.

**Mixed sequencer** This component perform the exact same things as the I^2C version. Except that the sequences it can execute must generate \texttt{mixed_req_item} transactions and receive \texttt{mixed_rsp_item} transaction as responses.

### 3.2.5.4 Mixed signal analysis component

The mixed analysis components are designed to gather mixed transactions from their matching agents (the PIO analysis component is receiving transaction from the PIO agent, ...) and also I^2C requests from the I^2C agent. The figure 3.9 shows the internal components and connections of each mixed analysis component in the testbench.

Just like what we did for the mixed agents, the mixed analysis components are also OVM components that are created from classes that derive from a custom base class: \texttt{mixed_analysis}. Here is the declaration of that class:

```overture
class mixed_analysis #(type adapter_type = mixed_adapter,
                    type scoreboard_type = mixed_scoreboard)
  extends ovm_component;
endclass
```

To design an analysis component that can be instantiated, we need to do like the following example does, which is the implementation of the PIO analysis component:

```overture
class pio_analysis extends mixed_analysis #(mixed_adapter, pio_scoreboard);

  'ovm_component_utils(pio_analysis)

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

endclass
```

As we can see, the \texttt{mixed_analysis} is responsible for instantiating the TLM ports of the analysis component, fetching configuration items and instantiate internal components accordingly. The specialized class has almost nothing but calling the constructor and register the class with the OVM registry (so it can be managed by the factory).

The rest of this section focuses on the base classes defined to created specialized scoreboard and adapter.
Figure 3.9: I²C PIO, DIG and AC analysis component as implemented in the default environment
### 3.2 OVM testbench

**Mixed adapter**  The `mixed_adapter` class actually does not need to be specialized. It can work as is. This class is akin to the I\(^2\)C adapter, except that it is dealing with mixed transactions, not I\(^2\)C transactions.

The adapter is receiving a stream of `mixed_base_item` transactions, which are in reality created from the `mixed_req_item` class. We are here taking advantages of the polymorphism of the mixed requests allowing them to be transferred via `mixed_base_item` specialized TLM ports\(^1\). The adapter is converting back those transactions to `mixed_req_item` and send them directly to the reference model in no simulation time step, assuring that the reference model can be stimulated at the same time as the DUT.

The reference model is sending back `mixed_base_item` transactions which model the events that the mixed monitor should be outputting. Those are directly sent to the scoreboard through an analysis port. The thing that is different from the I\(^2\)C adapter is that a request is not bounded to any response from the reference model.

**Mixed scoreboard**  The `mixed_scoreboard` is providing a basic implementation of a scoreboard to compare transactions from the monitor and the reference model. Let’s take an example, that will ease the explanation.

```pascal
class dig_scoreboard extends mixed_scoreboard;

  `ovm_component_utils(dig_scoreboard);

  function new(string name, ovm_component parent,
               string _fifos_id [] = '("d0","d1","d2","d3","ready");
      super.new(name, parent, _fifos_id);
  endfunction

endclass
```

Here is the `dig_scoreboard` class is a specialization of the base scoreboard. The idea here was to provide a base class that contains already an implementation of the `run()` task so the user can design a new scoreboard for a set of DUT pins only by specifying their name, as done in the example.

The scoreboard is receiving all the transactions coming out of the monitor into a single FIFO. A process of the `run()` task is sorting that FIFO according to the pin

---

\(^1\)We wanted to avoid redesigning a mixed talker for `mixed_req_item` transactions to be connected to the output stream of transactions from the driver. That’s also why the mixed drivers are outputting `mixed_base_item` transactions, even if they are `mixed_req_item` transactions in reality.
name, this means it is fetching a transaction from the input FIFO, read the associated pin name and send it to another FIFO corresponding to that pin. This is why the specialization needs to provide a list of pin names that are to be analyzed with the scoreboard.

The other input port is the one receiving the transactions from the reference model, via the adapter. All those transactions are also buffered into a input FIFO. Note that the reference model should only output a new transaction when a pin value of the DUT is supposed to change, in other words, for a same pin, it cannot generate two or more identical transactions in a row.

Another process of the run() task is responsible to compare transactions from the reference model and from the DUT. First it fetches a transaction from the input reference FIFO, then read the pin name within the transaction and fetch the one from the DUT using the corresponding FIFO and compares them.

A problem that may appear is small differences between the two transactions when they are carrying analog values. For example when the driver set the pin AC2 with a value of 3V, the reference model will be aware of that request and will generated a transaction for the scoreboard saying that the pin AC2 should change to 3V. However, due to the analog package of Semtech, the pin voltage will be resolved to a value that might not be exactly 3V since it is represented with a real value. This is why the comp() function coming along with mixed_base_item class is asking for a tolerance value when two analog values are compared.

Another problem that may also appear is that for one single transaction of the reference model, may correspond to multiple transactions from the DUT, for a single pin. The simulations showed that the output analog value of a DUT pin is not stable and might change slightly, a few microvolts by a few microvolts. This is however enough for the monitor to generated multiple transactions. In the end, if the reference model says that the next expected value for AC2 is 3V, the FIFO associated with that pin that gathers the transactions from the monitor may have multiple transactions with values very close to 3V. So, fetching that FIFO should be done smartly to consider all those transactions as one.

Finally the scoreboard is instantiated using the OVM factory, so anyone could
3.2 OVM testbench

design a new class deriving from `dig_scoreboard`, specialize it for a particular task by
re-implementing the `run()` task, and/or, by using the I²C request input port to detect
the activation of a particular feature of the DUT.

3.2.6 Using the testbenches

3.2.6.1 Library of sequences

We explained before that stimulus generators in an OVM testbench are sequencers
executing sequences upon the request of a test. It is required to design a number of
simple sequences covering the generation of all possible transactions. Those simple
transactions can be seen as a low level API to operate the DUT.

We have designed two sets of low level sequences:

**Mixed sequences** Those sequences are used to generate mixed request transactions.

The following piece of code shows a simple example of transaction creation by
the mean of a sequence. The `mixed_seq_set_bit` class presented here below has
the purpose of creating a `mixed_req_item` request transaction in order to write a
digital value to a given DUT input port.

```plaintext
class mixed_seq_set_bit extends ovm_sequence #(mixed_req_item, mixed_rsp_item);

    rand bit val;
    string id;

    task body();
        mixed_req_item pin;
        assert($cast(pin, create_item(mixed_req_item::get_type(),
            m_sequencer, id)));
        pin.set_values(SEND, DIGITAL, id, 0.0, 0.0, 0);
        start_item(pin);
        pin.val_dig = val;
        finish_item(pin);
    endtask
endclass
```

The request purpose is thus automatically set to `SEND` and the type set to `DIGITAL`. What is interesting here is the way the process that instantiating that
sequence is using to pass the value to be written and the targeted pin name.
This is done using the object variables of the class. Once a sequence is created out of this class, the variable `val` and `id` must be set from the outside, either by accessing directly the variable or using the constructor (not showed in the previous code). The other interesting thing is that we can use the `rand` keyword in front of any object variable allowing it to be randomized as described in the SystemVerilog LRM. This is how we are able to generate randomized transactions with or without specifying constraints.

**I2C sequences** I2C low level sequences are somehow more interesting to study regarding the randomization part. The following piece of code show the object variables of the `i2c_seq_read` sequence class. This class allow the programmer to instantiate a sequence to read some data from the I2C bus. When executed, the sequence generate a start bit, send the first byte with a slave address and start reading a given number of bytes provided by that slave. The stop bit is not emitted, allowing to execute another sequence directly, thus simulating a repeated start bit.

```vivado
class i2c_seq_read extends ovm_sequence #(i2c_req_item, i2c_rsp_item);
  ...
  // Inputs
  rand logic [6:0] slave_address;
  rand int unsigned data_size;
  // Outputs / results
  logic [7:0] data[];
  bit mismatch, ack, valid;
  int unsigned bit_read;

  constraint c_data_size {
    data_size inside {[1:10]};
  }
  ...
endclass
```

The slave address and the amount of byte to be read can be generated randomly within some constraints. Here the number of bytes to read is limited between 1 and 10, but the test that instantiate a sequence can add some in-line constraints before randomizing it. Next to the input variables, we have the output variables that are filled during the execution of the sequence, allowing to have a feedback.
3.2 OVM testbench

Do we have lost the arbitration process? Are the data valid? And of course the read bytes.

Actually, there is one missing type of low level sequences. We should add sequences to generate erroneous I²C communications because the I²C transactions have been designed to do so.

Concerning higher level sequences, they are simply reusing those low level sequences to generate more interesting stimulus. One example, to keep up with the randomization explained a few lines earlier, is the class called `i2c_seq_rand_rw`. It can be used to create sequences that will randomly read or write registers from the SX8723. Here is the `body()` task of that class:

```verbatim
task body();
    i2c_seq_sx_send seq_send;
    i2c_seq_sx_read seq_read;

    for (int unsigned i = 0; i < iterations ; i++) begin
        case ($random & 1)
            0: begin
                assert($cast(seq_send,
                             create_item(i2c_seq_sx_send::get_type(),
                                          m_sequencer, "send_data"));

                start_item(seq_send);
                assert(seq_send.randomize());
                finish_item(seq_send);
                end
            1: begin
                assert($cast(seq_read,
                             create_item(i2c_seq_sx_read::get_type(),
                                          m_sequencer, "read_data"));

                start_item(seq_read);
                assert(seq_read.randomize());
                finish_item(seq_read);
                end
        endcase
    end
endtask
```

We can clearly see how lower level sequences are created (using the factory) and randomized before been executed.
3.2 OVM testbench

3.2.6.2 Tests

The tests are used to reconfigure the default environment and execute one or many sequences. They are highly chip specific and should be designed and created according to a previously established verification plan. In the case of the SX8723, the verification plan was only planning directed tests since the old VHDL testbench was not able to handle randomization.

As examples, we reimplemented two directed tests from the old verification plan and designed a simple yet complete test showing how to trigger the execution of a randomizing sequence.

We will go through the most interesting designed test because it features all the steps required to configure the OVM environment and shows how to run a sequence with a particular sequencer from the testbench. The following test enables a special feature of the SX8723 that connect two internal clocks to the pin d0 and d1 after having written a special value in a register of the chip. The goal is to measure the frequency of those clocks and compare them with expected values. This is one of the directed test designed by Semtech prior to this project.

Let’s first focus on the build() function:

```plaintext
function void build();
    super.build();
    // Disable the monitor in the ac_agent
    set_config_int ("env.ac_agent", "has_talker", 0);
    set_config_int ("env.ac_agent", "has_monitor", 0);
    // Select a specialized scoreboard.
    dig_scoreboard::type_id::set_type_override(dig_scoreboard_ck_clocks::get_type());
endfunction
```

This code is simply using the config facility to tune the configuration of the AC agent. It is actually disabling the AC monitor and its associated talker because the AC agent is only responsible for analog inputs. Monitoring them isn’t useful and instantiating the monitor would just be a waste of processing power during the simulation. The second part of the build() function is the replacement of the generic scoreboard explained in the section 3.2.5.4 by a more specific one. The new one is actually analyzing the I^2C requests for the particular series of requests that enables the internal
3.2 OVM testbench

clocks to be connected to the output pins. Once this is detected, the scoreboard is able to measure their frequency, and to compare them with expected values. We can see here that the generic scoreboard, which was simply comparing transactions, is totally replaced by another scoreboard without having to modify the code of the testbench. We can tune this directly from the test class in a single line of code classes thanks to the OVM factory.

The next function to be executed by the core of OVM is `connect()`

```plaintext
function void connect();
    super.connect();
    //Connect the dig_analysis component with a feed of i2c_requests
    e.m_i2c_agent.driver_req_analysis_port[0]
        .connect(e.m_dig_analysis.i2c_analysis_export);

    // Select wether you want the analysis bloc to be connected to an
    // i2c_driver or to the i2c_monitor
    e.m_i2c_agent.driver_analysis_port[0]
        .connect(e.m_i2c_analysis.analysis_export);

    //e.m_i2c_agent.analysis_port.connect(e.m_i2c_analysis.analysis_export);
endfunction
```

This function is simply connecting the DIG analysis component (where resides the specialized scoreboard) to a feed of I²C requests. This has to be done manually by the test because the I²C agent can be configured to have multiple drivers and thus multiple output ports providing I²C requests. The same thing has to be done to connect the I²C analysis component with a source of i2c_pair_item transactions that are generated by the I²C monitor or any I²C driver.

Finally, the `run()` tasks which configure and run two types of sequence. The first one, `set_power`, is used to setup the power pins `vdd` and `vss`. Once configured, that sequence is associated with a sequencer from the PIO agent for execution. Same story for the second sequence `send_data`, except that it is used to write a register of the SX8723 with a special value using the I²C bus (that is what will trigger the internal clocks to be output). This is why this sequence has to be executed by the an I²C driver in the I²C agent.

```plaintext
 task run ();
    pio_seq_set_power set_power;
    i2c_seq_sx_send send_data;

    // configure power supply
```
3.2 OVM testbench

set_power=new();
set_power.vdd_int = 90;
set_power.vss_int = 0;

// configure the data to be written in a given register
// enabling the clocks to be connected to d0 and d1
send_data=new();
send_data.slave_address = SLAVE_ADDRESS;
send_data.register = REG_SRC;
send_data.data = 8'h05;

ovm_report_info ("run", "Start ck_clocks test");
set_power.start(e.m_pio_agent.sequencer);
#40us;
send_data.start(e.m_i2c_agent.sequencer[0]);
// wait some time allowing the score to measure the frequency
#100us;
global_stop_request();
ovm_report_info ("run", "End ck_clocks test");
endtask
3.2 OVM testbench
Chapter 4

Further discussions

4.1 Simulation including timings

The DUT that is been tested with the testbench is an RTL model with some grafted analog behaviors. However, we may want to be able to reuse the same testbench and the same developed tools to simulate a model of the SX8723 including timings (for example a gate-level netlist back annotated with a SDF description).

To be able to handle those models we need to provide tools for the scoreboard to correctly identify an event on the DUT, which can be difficult in this case because of the analog values that the testbench is required to handle. While the scoreboard will output a single transaction, representing an idealized event that must happen on a port, the monitor in the agent might generate multiple transactions that may contain very different values. The scoreboard should be able to identify all those transactions as a single event on the DUT port and finally compare it with the one from the reference model.

Let’s take a simple example, an analog pin of the DUT should see a voltage change to 3V. The reference model will simply generate a single transaction containing the analog value 3. Meanwhile the real DUT pin can change as shown in the figure 4.1. This means the monitor will output multiple transactions for each variation of the pin voltage. All those transactions are finally being sent to the scoreboard.

To associate all those transactions to a single event, represented by a single transaction generated by the reference model, the scoreboard must first buffer all the transactions. Doing so allow the reference model to generated its transactions independently
4.1 Simulation including timings

Figure 4.1: Voltage change on a single port of the DUT

from the timing of the DUT. It can send transactions to the scoreboard without taking care of when the corresponding event should happen on the DUT. The reference model store in the scoreboard input FIFO the events that will happen. In parallel, all the transactions generated by the monitor for a single DUT port are also stored into a FIFO (one FIFO per DUT port).

All this allow the scoreboard to analyze the transactions and compare them in a smart way. First, the scoreboard should get a transaction from the reference FIFOs, describing an event that must happen on a particular DUT port. With this information, it is able to analyze and detect the transactions coming from the DUT corresponding to the expected event.

This detection should first discard all the DUT transactions within a given simulation time span, because when a voltage change occurs on a pin, we might experience voltage overshoot and the analog value of those transactions might be completely different from the one expected, like showed on the figure 4.1. To do so, the transactions are required to carry the simulation time at which it has been generated by the monitor. Once all the transactions within this time span are discarded, the following transactions coming from the DUT must carry an analog value close enough to the reference value.

The actual testbench already implements all the required tools to do so:
4.2 Reusability of the developed components

- The transactions are buffered at their entrance in the scoreboard.
- The transactions from the DUT can be sorted according to their pin origin. A task in the mixed_scoreboard can be fork to do this automatically.
- The comp() function from the mixed_base_item allow to compare two analog transactions with a given tolerance.
- The mixed_base_item transactions coming out of the mixed monitors have a variable containing the simulation time at which they were created.

So, one could design scoreboards to replace the generic ones (using the factory) that suit timing based simulation.

4.2 Reusability of the developed components

OVM has been designed to provide testbench creators the ability to create versatile and reusable components. The combination of the OVM factory, the config facility and object oriented features of SystemVerilog helped us to design classes that can be used for different purposes in a project and also allows to reuse them from project to project.

A simple example of reusability, yet demonstrative, is used in the conducted work. Semtech did not had a TLM reference model for their SX8723 chip and the time allowed for this project was to short to design a complete one from scratch. In order to design a complete architecture and prove the concept, we decided to instantiate the SX8723 model a second time to play the role of the reference model. In order to make it compatible with OVM and able to use the TLM paradigm, we reused the I²C agent and mixed agents. The agents were configured to contain only a single driver (no monitor nor talker) which allowed us to create a fake reference model capable of receiving and emitting transactions, like showed in the testbench overview on the figure 3.2.

What’s more, the I²C agent along with the I²C SystemVerilog interface are completely chip independent. So they can be reused in another testbench for a completely different DUT which uses an I²C bus. The analysis part can also be reused to a certain extend or modified using the factory.
4.3 What about assertion based verification?

Same story for the **mixed** classes. If another model is using the **analog** type for its port (like many other chips in Semtech’s office in Neuchâtel), those classes can be reused to create new agents and analysis components in a short amount of time. They were designed to be independent from the pins they must handle. In our case, we split the pins of the SX8723 into three parts: AC, PIO, and DIG. One could do the same but for another chip.

**4.3 What about assertion based verification?**

The project intended to design a new testbench architecture for a medium complexity chip. Basically, the newly developed testbench has the following broad features:

**Stimulus Generation** This was done thanks to the sequences that can be used to design directed or constrained random tests.

**Functional coverage** By analyzing the stimulus generated, we are able to determine which feature of the DUT has been exercised.

**Built-in checks** Scoreboard are used to compare the responses of the DUT with expected values to validate the design.

One limitation of this testbench is that it ”simply” wraps around the DUT, and does not take into consideration the data flow within the DUT. The latter is considered as a black box. We are sending data on one side and looking at what is going on on the other side. The problem is, if, unfortunately, the scoreboard detects an error, how can we backtrace the error, find the stimulus (or set of stimulus) that caused it? With our testbench, all this should be done in the scoreboard by keeping track of the stimulus. For complex design, this can be very though to design and checking the waveform manually is prone to human errors.

To help with this problem, SystemVerilog provide assertion mechanisms allowing to plug in the design some small monitors directly at the signal level. This subject was already studied for the SX8723 in another master’s thesis (Stanic (2007)). Using assertions is complementary to a classic testbench and allow to easily perform protocol checking and small data checking directly inside the DUT. While our testbench tries to
take some distance from the DUT signals in order to generate interesting and various
stimulus, assertions keep up with the DUT signals and are able to detect an error
as soon as it appears (assuming there is enough of them plugged inside). This is an
important tool when the testbench is being designed after the RTL model (just like
in our case). It can also improve the simulation speed because there is no need to
backtrace everything in the scoreboard.
4.3 What about assertion based verification?
Chapter 5

Conclusions

5.1 Comparison between the old VHDL and the new OVM testbench

To compare easily the two testbenches, let’s sum up their main features. For Semtech’s VHDL testbench:

- Developed in the same language as the DUT (VHDL).
- Separated tests (*.drv files) and testbench.
- Static environment. It is required to modify the code of the bench_driver to modify the testbench environment.
- Directed tests only. The stimulus generation capabilities are limited by the set of commands the bench_driver is able to understand.
- The analysis of the the DUT responses is done after the simulation. Only a few simple checks can be done during the simulation.
- No functional coverage, because there is no stimulus randomization.

As for the new OVM testbench:

- Developed using SystemVerilog and the OVM methodology. The wrappers we were required to create lowered a bit the capabilities of the VHDL package, mainly due to Questasim limitations.
5.1 Comparison between the old VHDL and the new OVM testbench

- Separated tests and testbenches. Both are created out of SystemVerilog classes.
- Configurable environment. Each test can modify it according to its needs.
- Directed and constrained random tests. The concept of sequences allow us to higher the level of abstraction as much as we want, giving the possibility to create complex stimulus generators.
- The analysis is done on-the-fly, during the simulation. It however requires a TLM model of the DUT to generate expected responses.
- Functional coverage capabilities are inherent to the SystemVerilog language. OVM simply helps collecting the data.

Finally, the OVM testbench is able to handle all the previously designed directed tests, provides more flexibility and offers new tools for stimulus generation and responses analysis. However, all those improvements have a price. It usually takes longer to develop an OVM testbench and the simulation time can be increased for directed tests.

To quantify this increase of the simulation time, we run the same directed test on both testbenches. This test is applying two sine waves at the differential input of the ADC and the chip is configured to perform analog to digital conversion on those inputs continuously. The conversion results, when available, are read from the chip registers using the I²C bus. A waveform of the running test is showed in the figure B.1, in the appendix. For 3.4 simulated seconds, the simulator took 2639 seconds with the VHDL testbench and 3880 seconds with the OVM one (CPU time). We thus have, in this case, a performance drop of 32%. But do not forget that our OVM implementation of the testbench is using the RTL model twice: once for the DUT and once for the reference model. Using a real TLM model as the reference model can really improve the actual testbench performances knowing that it is actually more a proof of concept because it is not capable of find any bugs in the design.
5.2 Final note

The main objective of this project was to develop a new testbench architecture based on OVM using a DUT coming from the industry, which helped us to go one step further than what is usually presented in the literature about OVM. The final testbench is showing an example of what can be done using OVM on a real life device. It can be used by Semtech as a learning tool and to experiment new verification approaches before developing another OVM testbench for a project that is currently under development.

Unlike all the examples we may find about OVM testbenches, the SX8723 was a mixed-signal device and the simulation model is using a non standard VHDL package to handle analog behaviors. We successfully implemented wrappers for the DUT to make possible the communication between a SystemVerilog testbench and the VHDL design under test. Thanks to Questasim capabilities, this cohabitation can be done smoothly without creating uncommon workaround, event if we had to use some tricks to be able to use the analog package almost at its full capacity.

All the architecture is in place and running, however, the actual testbench for the SX8723 is still missing some OVM components to really start debugging the chip. No reference model has been developed nor complex scoreboards and coverage components, due mainly to time allowed for this project. Still, the most important parts are ready and all the components that can be reused amongst different projects are operational. The remaining work is mainly involving developing chip specific OVM components, tests and a complete verification plan suited for constrained random approaches. The time required to create those elements might be better spent for an upcoming product.

Julien Ghaye
Lausanne, june 19th, 2009.
Appendix A

Selected code source

A.1 Top level module

```plaintext
'timescale 1ns / 1ps

'include "chip_param.svh"

import ovm_pkg::*;

// Misc
import env_pkg::*;
import tests_pkg::*;
import virtual_interfaces_pkg::*;

// Top Level
module fullchip_tb ();

    // Dut
    i2c_if m_i2c_if();
    chip_if m_chip_if (.m_i2c_if(m_i2c_if));
    sx8723_wrapper dut (m_chip_if.dut);

    chip_vif chip_vif_obj;
    i2c_vif i2c_vif_obj;

    // Reference (Fake, just for testing)
    i2c_if ref_i2c_if();
    chip_if ref_chip_if (.m_i2c_if(ref_i2c_if));
    sx8723_wrapper reference (ref_chip_if.dut);

    chip_vif ref_chip_vif_obj;
    i2c_vif ref_i2c_vif_obj;
```

A.1 Top level module

```verbatim
// -------------------------------------
initial begin
  // DUT setup
  chip_vif_obj = new(m_chip_if);
  set_config_object("*.env.*", "pio_vif", chip_vif_obj, 0);
  set_config_object("*.env.*", "ac_vif", chip_vif_obj, 0);
  set_config_object("*.env.*", "dig_vif", chip_vif_obj, 0);
  i2c_vif_obj = new(m_i2c_if);
  set_config_object("*.env.*", "i2c_vif", i2c_vif_obj, 0);

  // Fake ref setup
  ref_chip_vif_obj = new(ref_chip_if);
  set_config_object("*.env_ref_tlm.*", "pio_vif", ref_chip_vif_obj, 0);
  set_config_object("*.env_ref_tlm.*", "ac_vif", ref_chip_vif_obj, 0);
  set_config_object("*.env_ref_tlm.*", "dig_vif", ref_chip_vif_obj, 0);
  ref_i2c_vif_obj = new(ref_i2c_if);
  set_config_object("*.env_ref_tlm.*", "i2c_vif", ref_i2c_vif_obj, 0);

  // Let's do it!
  run_test();
end
endmodule
```
A.2 VHDL wrapper

-- VHDL wrapper for the SX8723
--
-- Each pin of the chip (xemics.analog.analog type) is mapped as:
-- - an input port (analog_sv type), for writing values.
-- - an output port (analog_svtype), for reading values.
--
-- The basic usage of this wrapper is to write an analog_sv value to an input
-- pin (please refer to the analog_wrapper vhdl package). This value is
-- transferred to the DUT corresponding pin where the xemics library runs the
-- resolve function. Then the result is reported on the corresponding output
-- pin, ready to be read.
--
-- This wrapper also implements the pullup resistors of the I2C bus, since they
-- require the xemics.analog resolve function to be effective.
--
-- Compilation note: This file should be compiled with the -mixedsvvh to allow
-- Questasim to correctly use mixed language features.

-- Libraries

library ieee, xemics, lib_xe_project;
use ieee.std_logic_1164.all;
use xemics.analog.all; -- analog type
use work.analog_wrapper.all; -- analog_sv type

-- Entity

entity sx8723_vhdl is
  port(
    -- Input ports
    ac2_in  : in analog_sv;
    ac3_in  : in analog_sv;
    ac4_in  : in analog_sv;
    ac5_in  : in analog_sv;
    ac6_in  : in analog_sv;
    ac7_in  : in analog_sv;
    d0_in   : in analog_sv;
    d1_in   : in analog_sv;
    d2_in   : in analog_sv;
    d3_in   : in analog_sv;
  );
A.2 VHDL wrapper

```vhdl
-- Input ports
ready_in  : in analog_sv;
scl_in    : in analog_sv;
sda_in    : in analog_sv;
vdd_in    : in analog_sv;
vpump_in  : in analog_sv;
vss_in    : in analog_sv;

-- Output ports
ac2_out   : out analog_sv;
ac3_out   : out analog_sv;
ac4_out   : out analog_sv;
ac5_out   : out analog_sv;
ac6_out   : out analog_sv;
ac7_out   : out analog_sv;
d0_out    : out analog_sv;
d1_out    : out analog_sv;
d2_out    : out analog_sv;
d3_out    : out analog_sv;
ready_out : out analog_sv;
scl_out   : out analog_sv;
sda_out   : out analog_sv;
vdd_out   : out analog_sv;
vpump_out : out analog_sv;
vss_out   : out analog_sv;
```

end entity;

-------------------------------------------------------------------------------
-- Architecture
-------------------------------------------------------------------------------

architecture str of sx8723_vhdl is

-- Procedure simply converting the xemics.analog type to the analog_sv type
-- (and vice versa)

procedure analogsv2analog (signal ana_in: in analog; signal ana_out: out analog) is

variable temp: analog;
begin
    temp.v_value := ana_in.v_value;
temp.i_value := ana_in.i_value;
temp.z_value := ana_in.z_value;
temp.model := ana_in.model;
temp.device := ana_in.device;
temp.nb_gene := ana_in.nb_gene;
temp.nb_load := ana_in.nb_load;

```

end architecture str;

```
A.2 VHDL wrapper

```vhdl
ana_out <= temp;
end procedure;

procedure analog2analogsv (signal ana_in: in analog;
signal ana_out: out analog_sv) is
  variable temp: analog_sv;
begin
  temp.v_value := ana_in.v_value;
  temp.i_value := ana_in.i_value;
  temp.z_value := ana_in.z_value;
  temp.model := ana_in.model;
  temp.device := ana_in.device;
  temp.nb_gene := ana_in.nb_gene;
  temp.nb_load := ana_in.nb_load;
  ana_out <= temp;
end procedure;

-- Internal signals
signal ac2_i, ac3_i, ac4_i, ac5_i, ac6_i, ac7_i, d0_i, d1_i, d2_i, d3_i,
       ready_i, scl_i, sda_i, vdd_i, vpump_i, vss_i : analog;
begin
  -- DUT instanciation
  dut: entity lib_xe_project.sx8723(structural)
  port map (
    pad_ac2  => ac2_i,
    pad_ac3  => ac3_i,
    pad_ac4  => ac4_i,
    pad_ac5  => ac5_i,
    pad_ac6  => ac6_i,
    pad_ac7  => ac7_i,
    pad_d0   => d0_i,
    pad_d1   => d1_i,
    pad_d2   => d2_i,
    pad_d3   => d3_i,
    pad_ready => ready_i,
    pad_scl  => scl_i,
    pad_sda  => sda_i,
    pad_vdd  => vdd_i,
    pad_vpump => vpump_i,
    pad_vss  => vss_i
  );

  -- SCL and SDA pullup resistors (50kohms)
  process (vdd_i)
  begin
    v_gene(sda_i, vdd_i.v_value, 50.0e3);
  end process;
end
```
v_gene(scl_i, vdd_i.v_value, 50.0e3);
end process;

-- Mapping between the xemics.analog and the analog_sv type
analogsv2analog(ac2_in, ac2_i);
analogsv2analog(ac2_i, ac2_out);
analogsv2analog(ac3_in, ac3_i);
analogsv2analog(ac3_i, ac3_out);
analogsv2analog(ac4_in, ac4_i);
analogsv2analog(ac4_i, ac4_out);
analogsv2analog(ac5_in, ac5_i);
analogsv2analog(ac5_i, ac5_out);
analogsv2analog(ac6_in, ac6_i);
analogsv2analog(ac6_i, ac6_out);
analogsv2analog(ac7_in, ac7_i);
analogsv2analog(ac7_i, ac7_out);
analogsv2analog(d0_in, d0_i);
analogsv2analog(d0_i, d0_out);
analogsv2analog(d1_in, d1_i);
analogsv2analog(d1_i, d1_out);
analogsv2analog(d2_in, d2_i);
analogsv2analog(d2_i, d2_out);
analogsv2analog(d3_in, d3_i);
analogsv2analog(d3_i, d3_out);
analogsv2analog(ready_in, ready_i);
analogsv2analog(ready_i, ready_out);
analogsv2analog(scl_in, scl_i);
analogsv2analog(scl_i, scl_out);
analogsv2analog(sda_in, sda_i);
analogsv2analog(sda_i, sda_out);
analogsv2analog(vdd_in, vdd_i);
analogsv2analog(vdd_i, vdd_out);
analogsv2analog(vpump_in, vpump_i);
analogsv2analog(vpump_i, vpump_out);
analogsv2analog(vss_in, vss_i);
analogsv2analog(vss_i, vss_out);
end architecture;
A.3 SystemVerilog wrapper

`timescale 1ns / 1ps

// System verilog wrapper for the DUT.
//
// Provides a SystemVerilog (SV) interface (cfs. LRM) for the DUT to ease the
// connection between the DUT (wrapper in this case) and the SV testbench.
/*****************************************************************************/

import analog::*;
import analog_wrapper::*;

module sx8723_wrapper (chip_if.dut pads);

// sx8723_vhdl wrapper instanciation
// Connect the wrapper ports to the SV interface
sx8723_vhdl dut (pads.ac_in[2], pads.ac_out[2], pads.ac_in[3], pads.ac_out[3], pads.ac_in[4], pads.ac_out[4], pads.ac_in[5], pads.ac_out[5], pads.ac_in[6], pads.ac_out[6], pads.ac_in[7], pads.ac_out[7], pads.d_in[0], pads.d_out[0], pads.d_in[1], pads.d_out[1], pads.d_in[2], pads.d_out[2], pads.d_in[3], pads.d_out[3], pads.ready_in, pads.ready_out, pads.scl_in, pads.scl_out, pads.sda_in, pads.sda_out, pads.vdd_in, pads.vdd_out, pads.vpump_in, pads.vpump_out, pads.vss_in, pads.vss_out);

endmodule
A.4 Chip interface

`timescale 1ns / 1ps

//------------------------------------------------------------------------------
// chip_if SystemVerilog interface.
//
// Provides connection means to the DUT directly using the analog_sv type and
// or using digital values.
//
// The DUT’s wrapper divide each pin of the chip into 2 analog_sv ports
// (1 input for writing and 1 output for reading). This SV interface actually
// acts as the input and output points to the DUT by containing the variables
// connected to the DUT’s ports.
// If required by the physical port’s type, The latter might have a couple of
// digital variables (1 for writing a digital value, one for reading a digital
// value), in addition the the analog_sv variables.
//
//------------------------------------------------------------------------------

import analog::*;
import analog_wrapper::*;
import analog_pkg::*;

interface chip_if (i2c_if.extern_mp m_i2c_if);
//note, this interface requires a connection to a i2c_if SV interface since the
//SX8723 implements the I2C protocol.

//------------------------------------------------------------------------------
// Internal signals
//------------------------------------------------------------------------------

// Analog_sv variables
// *_in  are used to write an analog_sv value to the DUT
// *_out are used to read an analog_sv value from the DUT
analog_sv ac_in [7:2] , ac_out[7:2];
analog_sv d_in [3:0] , d_out [3:0];
analog_sv ready_in , ready_out;
analog_sv scl_in , scl_out;
analog_sv sda_in , sda_out;
analog_sv vdd_in , vdd_out;
analog_sv vpump_in , vpump_out;
analog_sv vss_in , vss_out;

// logic (digital) variables
logic sda_out_dig = 1, sda_in_dig;
logic scl_out_dig = 1, scl_in_dig;
logic d_in_dig [3:0], d_out_dig[3:0];
logic ready_in_dig , ready_out_dig;
A.4 Chip interface

//***************************************************************
// Modports
// Defines which of the previous variables should be accessed by who and
// how (in? out? ...)
//***************************************************************
modport dut (ref ac_in, ac_out, d_in, d_out, ready_in, ready_out,
            scl_in, scl_out, sda_in, sda_out, vpump_in, vpump_out,
            vdd_in, vdd_out, vss_in, vss_out);
modport pio_driver (input vss_out, vdd_out, vpump_out,
                    output vss_in, vdd_in);
modport pio_monitor (input vss_out, vdd_out, vpump_out);
modport ac_driver (input ac_out, d_out,
                   output ac_in, d_in);
modport ac_monitor (input ac_out, d_out);
modport dig_driver (output d_in_dig, d_in, ready_in_dig,
                   input d_out_dig, d_out, ready_out_dig);
modport dig_monitor (input d_out_dig, d_out, ready_out_dig);

//***************************************************************
// Glue logic : I2C port
//***************************************************************

// sda_out (analog, from chip) ---> sda_out_dig (digital)
always @(sda_out.v_value, sda_out.device, vdd_out.v_value)
  ana2dig(sda_out_dig, sda_out, vdd_out);
// same with scl
always @(scl_out.v_value, scl_out.device, vdd_out.v_value)
  ana2dig(scl_out_dig, scl_out, vdd_out);

// sda_in_dig (digital, from tests) ---> sda_in (analog, to DUT)
always @(sda_in_dig, vdd_in.v_value, vss_in.v_value)
  i2c_dig2ana(sda_in, sda_in_dig, vdd_in, vss_in);
// same with scl
always @(scl_in_dig, vdd_in.v_value, vss_in.v_value)
  i2c_dig2ana(scl_in, scl_in_dig, vdd_in, vss_in);

// Map the sda/scl lines of the i2c_if SV interface to the local variable
// sda_in_dig and scl_in_dig
assign sda_in_dig = m_i2c_if.sda;
assign scl_in_dig = m_i2c_if.scl;

// Determine what is the output value of the sda pin of the DUT and write in
// in the sda's driver list.
always_comb begin
  if (sda_in_dig)
    m_i2c_if.sda_out"_dut_" = sda_out_dig;
  else if(sda_in_dig == 0)
    if (sda_out.nb_gene > sda_in.nb_gene + 1)
      m_i2c_if.sda_out"_dut_" = 0;
    else
      m_i2c_if.sda_out"_dut_" = 1;
end

// Same with the scl line.
always_comb begin
  if (scl_in_dig)
    m_i2c_if.scl_out"_dut_" = scl_out_dig;
  else if (scl_in_dig == 0)
    if (scl_out.nb_gene > scl_in.nb_gene + 1)
      m_i2c_if.scl_out"_dut_" = 0;
    else
      m_i2c_if.scl_out"_dut_" = 1;
end

// Glue logic : GPIO and ready pins

// d_out[] (analog, from chip) ---> d_out_dig[] (digital, to testbench)
// d_in_dig[] (digital, from testbench) ---> d_in[] (analog, to chip)
generate
  genvar I;
  for (I=0; I<4; I=I+1) begin
    always @(d_out[I].v_value, d_out[I].device, vdd_out.v_value)
      ana2dig(d_out_dig[I], d_out[I], vdd_out);
    always @(d_in_dig[I], vdd_in.v_value, vss_in.v_value)
      dig2ana(d_in[I], d_in_dig[I], vdd_in, vss_in);
  end
endgenerate

// same with the ready pin
always @(ready_out.v_value, ready_out.device, vdd_out.v_value)
  ana2dig(ready_out_dig, ready_out, vdd_out);
always @(ready_in.v_value, vdd_in.v_value, vss_in.v_value)
  dig2ana(ready_in, ready_in_dig, vdd_in, vss_in);
endinterface;
A.5 I2C interface

from timescale 1ns / 1ps

// I2C interface

// This interface implements the i2c protocol

// According to the Modports, the designs using this interface might have
// access to:
// - sda/scl line : the actual i2c bus lines
// - sda_out[string] : Associative array containing the different drivers for
//   the sda line.
// - scl_out[string] : same as sda_out but for the scl line

// Bus state (not as described in the i2c spec!)
typedef enum(FREE, FREE_HD, END_COMM, BUSY) i2c_bus_state;

interface i2c_if;

    // wires / variables / events
    // scl and sda are ANDed wires (could also be wires since the AND behavior
    // of the bus is explicitly implemented further)
    wand sda=1, scl=1;

    // sda line 'IF' there was only the DUT as the driver
    wand sda_out_dut = 1;

    // internal variables
    logic sda_int, scl_int;

    // Associative array containing the sda/scl drivers.
    // sda_out["_dut_"] is reserved for DUT usage
    // scl_out["_dut_"] is reserved for DUT usage
    logic sda_out[string] = '{"_dut_":1}, scl_out[string] = '{"_dut_":1};

    // Internal synchronisation
    i2c_bus_state bus_state;
    event start_bit_e, restart_bit_e, stop_bit_e, end_comm_e;
    event scl_start[string], scl_stop[string];
    bit scl_gate[string];

//-------------------------------------------
// Modport
//-------------------------------------------------------------------------
modport master_mp(
    input bus_state, start_bit_e, restart_bit_e, stop_bit_e
);
modport slave_mp(
    input bus_state, start_bit_e, restart_bit_e, stop_bit_e
);
modport monitor_mp(
    input bus_state, start_bit_e, restart_bit_e, stop_bit_e
);
modport extern_mp(
    inout sda, scl,
    output sda_out, scl_out
);

// Drive the sda/scl lines according to the drivers list sda_out[] and
// scl_out[] with some given rising and falling time
//-------------------------------------------------------------------------
parameter T_R = 1us,
    T_F = 0.3us;

// link logic <-> wand
always_comb begin
    if (sda_out.size() > 0) begin
        sda_int <= sda_out.and;
        scl_int <= scl_out.and;
    end else begin
        sda_int <= 1;
        scl_int <= 1;
    end
end

assign #(T_R, T_F) sda = sda_int;
assign #(T_R, T_F) sda_out_dut = sda_out["_dut_"];
assign #(T_R, T_F) scl = scl_int;

// Tasks
//-------------------------------------------------------------------------
// Those are low level task to operate the i2c bus lines. A design using
// this i2c SV interface should only use those tasks to access the bus,
// especially to drive new values on the lines since those tasks have been
// designed to work together smoothly. Mixing those and user-defined task
// might create unexpected behaviors on the bus.
//
// tasks:
A.5 I2C interface

// ------
// register (string id);
// send_bit_master (string id, bit b, output bit mismatch);
// send_bit (string id, bit b, output bit mismatch);
// read_bit_dut_master (output bit b, output bit invalid, input string
// read_bit_dut (output bit b, output bit invalid);
// read_bit_master (output bit b, output bit invalid, input string
// read_bit (output bit b, output bit invalid);
// scl_gen (string id);
// repeat_start_bit (string id);
// start_bit (string id);
// stop_bit (string id);

// Timing parameters as specified in the I2C spec.
parameter T_HD_STA = 4us,
T_SU_STA = 4.7us,
T_HD_DAT = 0us,
T_LOW = 4.7us,
T_HIGH = 4us,
T_SU_STO = 4.0us;

// register()
//
// This tasks must be instanciated by any designs which purpose is to write
// values on the i2c bus (masters, slaves transmitter). The design
// provides a 'id' which will be used to create associated sda/scl drivers
// within the i2c interface and other internal logic for clock generation
// (i2c master only)
task automatic register(string id);
    event e1;
    event e2;
    sda_out[id]=1;
    scl_out[id]=1;
    scl_start[id] = e1;
    scl_stop[id] = e2;
    scl_gate[id] = 0;
endtask

// scl_gen()
//
// This task forks a new process and returns immediately.
// The forked process generates the internal clock associated with each
// i2c master as specified in the I2C specification.
task automatic scl_gen (string id);
    fork
        // kill the clock generation if a stop bit is detected
        @(scl_stop[id]) disable ClkGen;
    endfork
begin : ClkGen
    // waits for a start bit to generate the clock
    @(scl_start[id]);
    wait(scl_gate[id]) scl_out[id] = 0;
    forever begin
        #T_LOW scl_out[id] = 1;
        @(posedge scl);
        fork
            begin : t_high
                #T_HIGH wait(scl_gate[id]) scl_out[id] = 0;
                @(negedge scl);
            end
        begin
            // check if another device on the scl line does not
            // force scl to 0
            @(negedge scl) disable t_high;
        end
end
    end
    join_any
endtask

// start_bit()
// Generate a start bit on the bus and start the process to generate scl
    task automatic start_bit(string id);
        // discard if bus is BUSY (may hold !!!)
        wait (bus_state == FREE || bus_state == FREE_HD);
        // if bus is still FREE (ie first master to issue a START)
        if (bus_state == FREE) begin
            sda_out[id] = 0;
            scl_gen(id);
            fork
                @(negedge scl) sda_out[id] = 1; //release sda at next negedge
            #T_HD_STA -> scl_start[id];
            join_any
        end
endtask

// repeat_start_bit()
// Generate a repeated start bit on the bus and start the process to generate scl
    task automatic repeat_start_bit(string id);
        scl_gate[id] = 1;
        -> end_comm_e;
        @(negedge scl);
#T_HD_DAT sda_out[id] = 1;
@(posedge scl);
-> scl_stop[id];
scl_gate[id] = 0;
#T_SU_STA sda_out[id] = 0;
scl_gen(id);

fork
   @(negedge scl) sda_out[id] = 1; //release sda at next negedge
#T_HD_STA -> scl_start[id];
join_any

endtask

// stop_bit()
/
// Generate a stop bit on the bus
task automatic stop_bit(string id);
scl_gate[id] = 1;
-> end_comm_e;
@(negedge scl);
#T_HD_DAT sda_out[id] = 0;
@(posedge scl);
-> scl_stop[id];
#T_SU_STO sda_out[id] = 1;
scl_gate[id] = 0;
endtask

// send_bit_master()
/
// Using its 'id', a master i2c component can read the sda bus line.
// This task has 2 main functions:
// - allow the i2c clock associated with the master 'id' to be generated
// - write the sda line with the value 'b'.
// If the write fail (i2c arbitration), the mismatch output port is set to
// 1'b1. This task returns after rising edge of the scl line.
task automatic send_bit_master(string id, bit b, output bit mismatch);
scl_gate[id] = 1;
send_bit(id, b, mismatch);
scl_gate[id] = 0;
endtask

// send_bit()
/
// same as send_bit_master() but without clock generation
// send_bit(string id, bit b, output bit mismatch);
// bit temp;
@
fork
A.5 I2C interface

```verilog
#T_HD_DAT sda_out[id] = b;
@negedge scl sda_out[id] = 1; // release the bus at next negedge
join_any

read_bit(temp, mismatch); // read written bit
mismatch = mismatch || !(temp == b); // mismatch is set to 1 if
  // a STOP/START will occur
  // arbitration process
endtask

// read_bit_dut_master()
//
// This task releases the scl generation and read the dut's sda value
// (what the dut wants to put on the line, not what is on the line).
// The task returns after the rising edge of scl. The output 'b' contains
// the read value. If 'invalid' is set to 1, it means a stop bit has been
// requested by another device and the 'b' value should not be considered.
//
// read_bit_dut_master(output bit b, output bit invalid, input string id);
//
// This task releases the scl generation and read the dut's sda value
// (what the dut wants to put on the line, not what is on the line).
// The task returns after the rising edge of scl. The output 'b' contains
// the read value. If 'invalid' is set to 1, it means a stop bit has been
// requested by another device and the 'b' value should not be considered.
//
// read_bit_dut_master(output bit b, output bit invalid, input string id);
//
// This task releases the scl generation and reads the sda line
// same as read_bit_dut_master() but reading the sda line
//
// read_bit_master(output bit b, output bit invalid, input string id);
//
// This task releases the scl generation and reads the sda line
// (what the dut wants to put on the line, not what is on the line).
// The task returns after the rising edge of scl. The output 'b' contains
// the read value. If 'invalid' is set to 1, it means a stop bit has been
// requested by another device and the 'b' value should not be considered.
//
// read_bit(output bit b, output bit invalid);
```
parameter T_HD_STA_MIN = 4us,
    T_BUF   = 4.7us;

// Sense bus state
always begin
    bus_state = FREE;
    @(start_bit_e);
    bus_state = FREE_HD;
    #T_HD_STA_MIN bus_state = BUSY;
    forever begin
        @(end_comm_e);
        bus_state = END_COMM;
        fork
            @(start_bit_e) bus_state = BUSY;
            @(stop_bit_e) #T_BUF bus_state = FREE;
        join_any
        disable fork;
        if (bus_state == FREE) break;
    end
end

// Sense start bit
always begin
    @(negedge sda);
    #0us;
    if (scl == 1) -> start_bit_e;
end

// Sense restart bit
always begin
    @(negedge sda);
    #0us;
    if (scl == 1 && bus_state == BUSY) -> restart_bit_e;
end

// Sense stop bit
always begin
    @(posedge sda);
    #0us;
    if (scl == 1) -> stop_bit_e;
end
endinterface
A.6  OVM environment

ifndef ENV_SVH
'define ENV_SVH

////////////////////////////////////////////////////////
// Env
////////////////////////////////////////////////////////
class env extends ovm_component;

//typedef ovm_object_registry #(env) type_id;
'ovm_component_param_utils(env)

localparam nbr_i2c_drivers = 1;

// Reference model
ref_tlm m_ref_tlm;

// Agents and analysis components
pio_agent m_pio_agent;
pio_analysis m_pio_analysis;
ac_agent m_ac_agent;
ac_analysis m_ac_analysis;
i2c_agent m_i2c_agent;
i2c_analysis m_i2c_analysis;
dig_agent m_dig_agent;
dig_analysis m_dig_analysis;

function new(string name, ovm_component parent = null);
   super.new(name, parent);
   $display("----------------------------------------------------------");
   $display("SX8723 OVM Testbench");
   $display("Company : Semtech");
   $display("Author : Julien Ghaye");
   $display("Date : June 2009");
   $display("----------------------------------------------------------");
endfunction

function void build();
   $display("-----START BUILD PHASE------------------------------------");
   //PIO config
   set_config_int ("pio_agent", "has_monitor", PIO_HAS_MONITOR);
   set_config_int ("pio_agent", "has_driver", PIO_HAS_DRIVER);
   set_config_int ("pio_agent", "has_sequencer", PIO_HAS_SEQUENCER);
endfunction
A.6 OVM environment

```verilog
set_config_int("pio_agent", "has_talker", PIO_HAS_TALKER);
set_config_int("pio_agent", "has_driver_talker", PIO_HAS_DRIVER_TALKER);

set_config_int("pio_analysis", "has_scoreboard", PIO_HAS_SCOREBOARD);
set_config_int("pio_analysis.scoreboard", "tolerance", $realtobits(PIO_TOLERANCE));

// AC config
//
set_config_int("ac_agent", "has_monitor", AC_HAS_MONITOR);
set_config_int("ac_agent", "has_driver", AC_HAS_DRIVER);
set_config_int("ac_agent", "has_sequencer", AC_HAS_SEQUENCER);
set_config_int("ac_agent", "has_talker", AC_HAS_TALKER);
set_config_int("ac_agent", "has_driver_talker", AC_HAS_DRIVER_TALKER);

set_config_int("ac_analysis", "has_scoreboard", AC_HAS_SCOREBOARD);
set_config_int("ac_analysis.scoreboard", "tolerance", $realtobits(AC_TOLERANCE));

// DIG config
//
set_config_int("dig_agent", "has_monitor", DIG_HAS_MONITOR);
set_config_int("dig_agent", "has_driver", DIG_HAS_DRIVER);
set_config_int("dig_agent", "has_sequencer", DIG_HAS_SEQUENCER);
set_config_int("dig_agent", "has_talker", DIG_HAS_TALKER);
set_config_int("dig_agent", "has_driver_talker", DIG_HAS_DRIVER_TALKER);

set_config_int("dig_analysis", "has_scoreboard", DIG_HAS_SCOREBOARD);
set_config_int("dig_analysis.scoreboard", "tolerance", $realtobits(DIG_TOLERANCE));

// I2C config
//
set_config_int("i2c_agent", "drivers", I2C_DRIVERS);
set_config_int("i2c_agent", "sequencers", I2C_SEQUENCERS);
set_config_int("i2c_agent", "has_monitor", I2C_HAS_MONITOR);
set_config_int("i2c_agent", "has_talker", I2C_HAS_TALKER);
set_config_int("i2c_agent", "has_driver_talker", I2C_HAS_DRIVER_TALKER);
for(int i = 0; i < I2C_DRIVERS; i++) begin
    string s;
    $sformat(s, "i2c_agent.driver_%0d", i);
    set_config_int(s, "read_dut", I2C_READ_DUT);
```

100
A.6 OVM environment

end

set_config_int("i2c_analysis", "has_coverage", I2C_HAS_COVERAGE);
set_config_int("i2c_analysis", "has_scoreboard", I2C_HAS_SCOREBOARD);

//
// component instanciation
//
m_pio_agent = new("pio_agent", this);
m_pio_analysis = pio_analysis::type_id::create("pio_analysis", this);
m_ac_agent = new("ac_agent", this);
m_ac_analysis = ac_analysis::type_id::create("ac_analysis", this);
m_i2c_agent = new("i2c_agent", this);
m_i2c_analysis = i2c_analysis::type_id::create("i2c_analysis", this);
m_dig_agent = new("dig_agent", this);
m_dig_analysis = dig_analysis::type_id::create("dig_analysis", this);
m_ref_tlm = new("ref_tlm", this);
endfunction

function void connect();
  m_pio_agent.driver_analysis_port.connect(m_pio_analysis.driver_analysis_export);
  m_pio_agent.analysis_port.connect(m_pio_analysis.analysis_export);
  m_ref_tlm.pio_put_port.connect(m_pio_analysis.put_export);
  m_ref_tlm.pio_get_port.connect(m_pio_analysis.get_export);

  m_ac_agent.driver_analysis_port.connect(m_ac_analysis.driver_analysis_export);
  m_ref_tlm.ac_put_port.connect(m_ac_analysis.put_export);
  m_ref_tlm.ac_get_port.connect(m_ac_analysis.get_export);

  m_i2c_agent.driver_req_analysis_port[0].connect(m_i2c_analysis.req_analysis_export);
  m_ref_tlm.i2c_put_port.connect(m_i2c_analysis.put_export);
  m_ref_tlm.i2c_get_port.connect(m_i2c_analysis.get_export);

  m_dig_agent.driver_analysis_port.connect(m_dig_analysis.driver_analysis_export);
  m_dig_agent.analysis_port.connect(m_dig_analysis.analysis_export);
  m_ref_tlm.dig_put_port.connect(m_dig_analysis.put_export);
  m_ref_tlm.dig_get_port.connect(m_dig_analysis.get_export);

  $display("-----END CONNECT PHASE------------------------------------");
endfunction

function void end_of_elaboration();
  $display("-----END ELABORATION PHASE--------------------------------");
endfunction

function void extract();
  $display("-----START EXTRACT - CHECK - REPORT PHASES----------------");
endfunction
endclass

'endif
Appendix B

Screenshots

B.1 ADC conversion waveforms

The figure B.1 presents an example of ADC conversion. Two sine wave are applied on the pin $ac2$ and $ac3$. The ZoomingADC$^\text{TM}$ in configured to select make the conversion on the differential value of those two inputs.

The right part of the figure shows a global waveform showing the configuration of the ZoomingADC$^\text{TM}$ via the I$^2$C bus. Later on the simulation, the conversion value is retrieved from the chip register each time the ready pin is activated.

The left part of the figure shows more detail about the I$^2$C communication to recover the converted value.
Figure B.1: Waveform of the ZoomingADC™ in action
### B.2 Functional coverage

**Figure B.2: Functional coverage after the execution of a constrained random test using Questasim**
References


SEMTECH (2007). *SX8724 ZoomingADC™ for Pressure and Temperature Sensing*.

