Verification Course
Exercises

June 2024

Contents

1 Introduction

The following document will provide to the reader an introduction to Universal Verification Methodology (UVM) verification through exercises to implement a testbench (TB) using several open-source tools.

The goal of these exercises is to:

- Understand the basic architecture of a UVM testbench using *PyUVM*;
- Understand the vertical reuse concept by integrating a Universal Verification Component (*uVC*) in a UVM testbench;
- Implement a library of tests and a library of virtual sequences, to fully verify the behavior of the Device Under Test (DUT);
- Implement UVM components, such as the Coverage and the Scoreboard, to collect testbench and design metrics.

For the exercises provided in document, it is assumed that the reader has received the source code of the UVM testbench, which is the intended template to support the exercises' development. In the document, the `<ROOT>` directory shall map to the base folder containing the source code received.

2 Design of the Saturation Filter

A *Saturation Filter* is a device that saturates the input when this exceeds certain limits. The design provided implements this device and will be used as the DUT (Device Under Test) for the following exercises. The device saturates the input data when this exceeds the threshold value defined.

2.1 Implementation

The figure 2.1 shows the high-level diagram of the *Saturation Filter*, showing the input and output ports of the design.

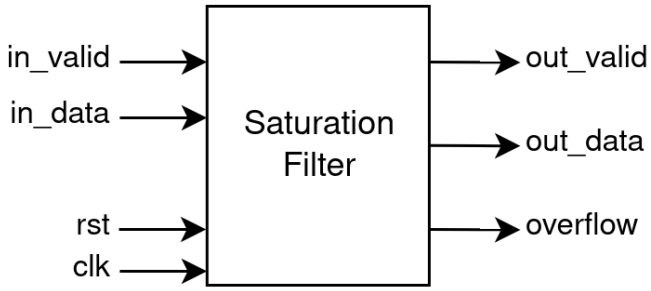

Fig. 2.1: *Saturation Filter* module.

The *Saturation Filter* I/O ports are the following:

- `clk`, the clock signal;
- `rst`, the reset signal;
- `in_valid`, the input valid signal;
- `in_data`, the input data sample;
- `out_valid`, the output valid signal;
- `out_data`, the output data sample;
- `overflow`, the output overflow signal.

The *Saturation Filter* module contains the following parameters that can be tuned to change the module operation:

- `DATA_W`, sets the data width of the data signal;
- `THRESHOLD`, the value over which to saturate the data.

The device uses the SyoSil Data Transfer Protocol (*sSDT*) 3, and was developed respecting the following requirements:

- **RS/01**: It shall be possible to reset the state of the saturation filter by toggling the `rst` input signal.
- **RS/02**: All signals shall react to the rising edge of the input `clk` signal.
- **RS/03**: The saturation functionality of the design shall be evaluated with the `THRESHOLD` parameter value. If the data is below the limits it is propagated to the output on the next rising edge of the `clk` signal. Otherwise, the data is `saturated` and the `out_data` signal will be the `THRESHOLD` value instead.
- **RS/04**: The `in_valid` signal is always propagated to the output on the next rising edge of the `clk`.
- **RS/05**: The *Saturation Filter* shall be compliant with the *sSDT* (simple SyoSil Data Transfer) protocol described in the following section.

3 Simple SyoSil Data Transfer Protocol

The simple SyoSil Data Transfer (*sSDT*) protocol is a simple synchronous data transfer protocol. The protocol has two signals, data and valid, and has two variants, the "producer" and the "consumer", which are modifying the direction of the signal as showed in the Table 3.1.

Signal Name	Producer direction	Consumer direction	Comment
valid	output	input	when asserted, data is valid
data	output	input	the data of the protocol

Table 3.1: *sSDT* protocol signals.

The protocol operation requires that the `data` signal must be valid when the `valid` signal is enabled. When the `valid` signal is disabled the `data` signal must be `0`. A timing diagram outlining the behavior of the protocol is shown in Figure 3.1.

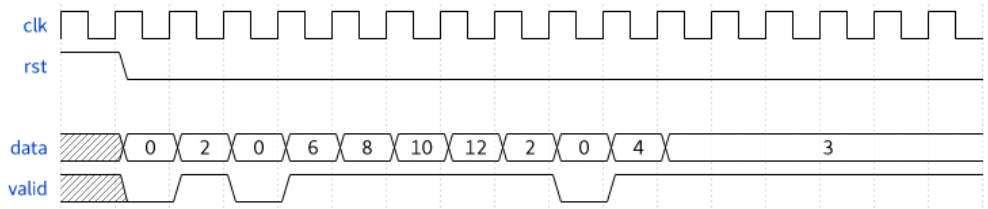

Fig. 3.1: *sSDT* protocol waveform.

4 *sSDT* Universal Verification Component

The *sSDT* Universal Verification Component (*uVC*) is implemented with the intention of generating traffic compliant with the protocol requirements. By developing the *sSDT uVC* it is ensured the reuse of the same component in multiple testbenches, without having to re-implement the code from the beginning. The Figure 4.1 shows the high-level diagram of the *sSDT uVC*.

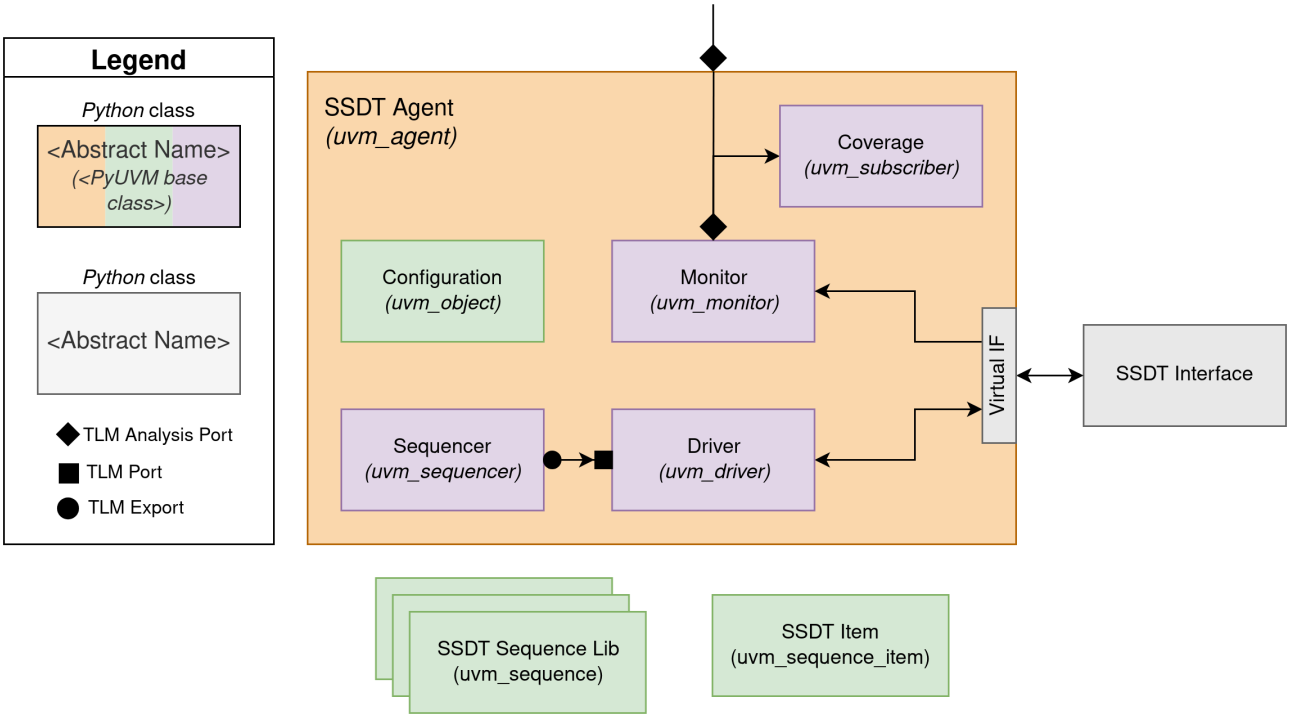

Fig. 4.1: *sSDT uVC* diagram.

The UVM components described in the Figure 4.1 can be found in the locations defined in the Table 4.2. Consider for the `<ROOT_sSDT>` the following path, `<ROOT>/sat_filter/src/tb/uvc/ssdt`/src.

Component	Path
SSDT Agent	`<ROOT_sSDT>/uvc_ssdt_agent.py`
Configuration	`<ROOT_sSDT>/uvc_ssdt_config.py`
Monitor	`<ROOT_sSDT>/uvc_ssdt_monitor.py`
Driver	`<ROOT_sSDT>/uvc_ssdt_driver.py`
Sequencer	`uvm_sequencer` base class
Coverage	`<ROOT_sSDT>/uvc_ssdt_coverage.py`
SSDT Interface	`<ROOT_sSDT>/uvc_ssdt_interface.py`
SSDT Sequence Lib	`<ROOT_sSDT>/uvc_ssdt_sequence_lib.py`
SSDT Item	`<ROOT_sSDT>/uvc_ssdt_seq_item.py`

Table 4.2: *sSDT* UVC files location.

5 Back2Back Testbench

The Back2Back testbench is developed with the intention of testing the Universal Verification Component (uVC) which is responsible for generating the traffic in compliance with the $sSDT$ protocol.

As showed in the figure 5.1, two $sSDT$ agents will be instantiated in the testbench and connected to the same interface. The "producer" shall drive the data on the interface, while the "consumer" will sample the content of the interface and broadcast them to the testbench.

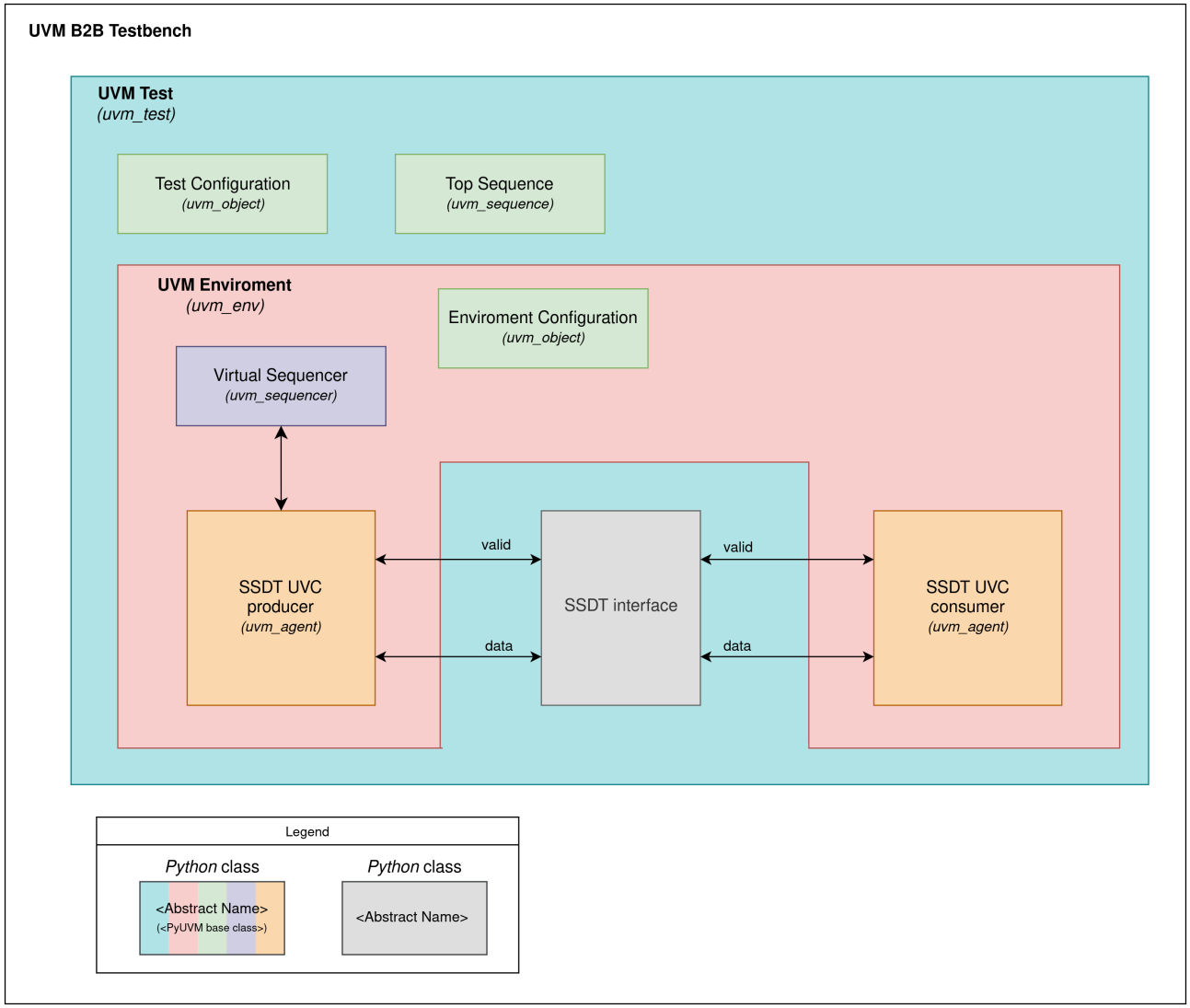

Fig. 5.1: UVM testbench for the back2back tests for the $sSDT$ uVC.

6 Generic Testbench Architecture

The Figure 6.1 shows the high level structure of the *Saturation Filter* UVM testbench. Analyzing the legend of the diagram it can be seen that most of the classes implemented and used in this testbench are UVM classes. All these base classes represent the "backbone" of the *PyUVM* library that aims to implement the UVM verification methodology in the Python programming language.

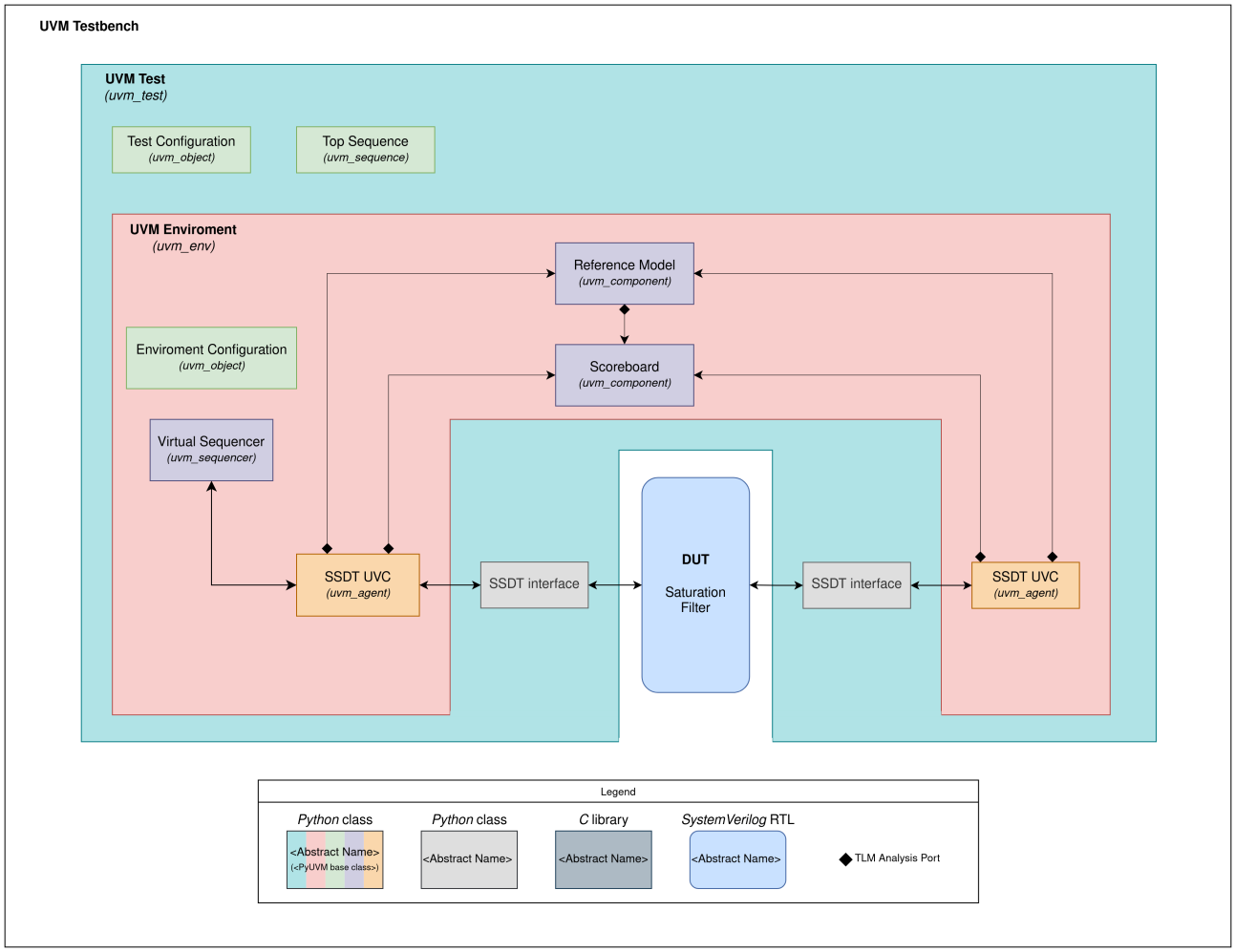

Fig. 6.1: UVM testbench for the *Saturation Filter*.

6.1 Base Test

All the testbench components are created and connected in the base test, which is an extension of the `uvm_test`. The test defines the handles for the environment class (`UVM Environment`), for the configuration object class (`Test Configuration`), the sequence class (`Top Sequence`), and for the interface classes required to communicate with the *uVC* (`\ssdt interface`).

6.1.1 Build Phase

In the `build_phase` all the class' instances are created. This is achieved with the create method which invokes the class constructor. The major benefit of using the create method is to register the class in the `UVM Factory`. The registration mechanism improves the efficiency and flexibility of the UVM testbenches by allowing the user to do instance overrides and checks for registered type. The interface class is not part of the *PyUVM* library, so the creation cannot be performed with the create method.

During the `build_phase` it is also possible to set the environment and the configuration object classes inside the configuration database (`ConfigDB`) to easily pass these class instances to the entire testbench.

6.1.2 Connect Phase

After all the components have been created, the base test shall connect them as necessary during the `connect_phase`. This implies the correct connection of the DUT signals to the two *sSDT* interfaces, one used by the producer and the other by the consumer.

6.1.3 Run Phase

In contrast with the other two functions mentioned above, the `build_phase` and `connect_phase`, which can be considered as regular Python functions, the `run_phase` on the other hand is a coroutine. This implies that the `run_phase` takes simulation time to run and is the only place where other coroutines can be launched. The `run_phase` is the main simulation phase of the `uvm_test` responsible for running configuration and data traffic sequences to the DUT.

6.2 Environment

The environment component, extended from `uvm_env`, defines handles for the configuration object, the virtual sequencer and the necessary *uVC*'s agents. In the scope of the *Saturation Filter* testbench, the environment contains one *sSDT uVC* acting as a producer and another *sSDT uVC* acting as a consumer. If there is a reference model, the environment shall define handles for the reference model handler and the scoreboard.

6.2.1 Build Phase

In the `build_phase` all the handles previously defined are created similarly as for base test (e.g. using the `create` method). Furthermore, the configuration object can be taken from the `ConfigDB`, through the `get` method, if it was already set by the base test. The configuration object can then be used to update the configuration for all agents inside the environment.

6.2.2 Connect Phase

The `connect_phase` is responsible for the correct connection of the instantiated components. The connection between the sequencers inside the *uVC* agents and the virtual sequencer is done by passing the appropriate handler. Another type of connection is the one between the *uVC* agents and the scoreboard, respectively with the reference model. This is based on a TLM interface, in which analysis port from the agents are connected with analysis export from the scoreboard and reference model.

6.3 Virtual Sequencer

The virtual sequencer defines handles only for the configuration object and the necessary *uVC*'s sequencers. The connection with other testbench components (e.g. agent's sequencers) is completed by the environment.

6.4 Test Configuration

The test configuration, extended from `uvm_object`, creates all *uVC*'s configuration objects. As was previously explained for the base test, the method create is called for all these components. For the *Saturation Filter* testbench, the `data_width` and `threshold` are the configurable parameters.

6.5 Reference Model

The reference model defines handles for the FIFOs that are connected to *uVC*'s analysis ports.

6.5.1 Build Phase

The FIFOs are created in the `build_phase` of the reference model calling the `uvm_tlm_analysis_fifo` constructor. In the case of the *Saturation Filter* testbench, two FIFOs were created, one for each *uVC*.

In addition to the FIFOs, the `build_phase` of the reference model also creates an analysis port for the communication with the scoreboard component. This is required for sending the reference model items the expected output for the comparison that takes place inside the scoreboard. A coroutine has been defined to make sure that the reference model receives the same inputs as the DUT and generates outputs that are defined to be correct.

6.5.2 Run Phase

The `run_phase` of the reference model is in charge with launching this coroutine.

6.6 Scoreboard

The scoreboard receives both the output from the DUT and from the reference model. The communication with the *uVC*'s and the reference model it is done using `TLM Analysis Ports`. In the *Saturation Filter* testbench the scoreboard defines handles for two FIFOs and two corresponding queues. The first pair is responsible for monitoring and sending the consumer items, while the second one is in charge with the reference model items.

6.6.1 Run Phase

In the `run_phase`, the scoreboard checks the content of the two queues, element by element. If a mismatch is found, an error is asserted.

7 Exercises

Exercise 0: Computer Setup

The following exercise will consist in setting up the Linux environment and install all the required simulation tools to compile the design and the testbench code, running simulations and visualizing the waveforms of all the signals involved.

The guide `client-setup.pdf` available in `<ROOT>/client_setup/docs/`, describes the steps to achieve this.

Exercise 1: *Saturation Filter* Walkthrough

Objective: Get familiarized with the existing testbench code and the commands to run tests.

Exercise: Analyze the *Saturation Filter* testbench source code.

Setup Python environment

```
[<username>@<servername> <ROOT>]$ source bin/venv.src.me
[<username>@<servername> <ROOT>]$ source .venv/bin/activate
# You are now in the Python Virtual environment
(.venv) [<username>@<servername> <ROOT>]$
```

Saturation Filter walkthrough

The following steps will give a walkthrough of the *Saturation Filter* UVM testbench:

1. Go to the testbench directory, `<ROOT>/sat_filter/src/tb`, and check the code for the *Saturation Filter* UVM testbench.

   Check section 6 for a more detailed description of the testbench structure.

2. Run the default test case to familiarize with the flow and the `GTKWave` tool for waveform visualization:

   2.1 On the terminal, run the following command to execute the available test:

   ```
   (.venv) [<username>@<servername> tb]$  make MODULE=test_sat_filter_default_seq
   ```

   where, `test_sat_filter_default_seq` is the name of the available test inside the tests folder, `<ROOT>/sat_filter/src/tb/tests`.

   2.2 Open the `GTKWave` application and load the waves:

   ```
   (.venv) [<username>@<servername> tb]$  gtkwave sim_build/<waveform-name>.vcd
   ```

   replace `<waveform-name>` by the generated waves from the previous step.

   More information related with the `GTKWave`, can be found in the GTKWave user-guide.

   2.3 Read the `Makefile` available in `<ROOT>/sat_filter/src/tb` to check how the DUT parameters are set.

   2.4 Try to rerun the test by running the following command and check the waves again:

   ```
   (.venv) [<username>@<servername> tb]$  make MODULE=test_sat_filter_default_seq \
   DATA_W=16 THRESHOLD=8
   ```

Exercise 2: *PyUVM* Introduction

Objective: Introduction to the *cocotb* framework, *PyUVM* library and UVM concepts.

Read the following pages in *Python for RTL Verification* by Ray Salemi to get familiar with Python and *cocotb* which will be explored more in details in the following days of the course.

- Introduction: p. 1-8
- Python Basics: p. 9-30
- *cocotb*: p. 143-176
- Why UVM?: p. 197-200

Exercise 3: *cocotb* Test

Objective: Introduction to the *cocotb* tests.

Exercise: Development of a *cocotb* test using the *Saturation Filter*.

Locate the *cocotb* exercise in `<ROOT>/exercises/sat_cocotb_exercise/`.

Open the `sat_default_test.py` and analyze the file. Then, run the test and visualize the waveforms, running the following commands:

```
# Run the tests
(.venv) [<username>@<servername> sat_cocotb_exercise]$ make

# visualize the waveforms
(.venv) [<username>@<servername> sat_cocotb_exercise]$ gtkwave \
sim_build/sat_filter_waves.vcd
```

Part 1: Random Test

Now, create a random test. Using the `sat_default_test.py` file as reference, create a *cocotb* test inside the `sat_random_test.py` file. The test should have the following specifications:

- Generate 10 sets of random input data;
- Ensure that the `in_valid` signal is high when setting the `in_data`;
- Assign the data to the input ports of the DUT (the *Saturation Filter*) on the rising edge;
- Wait at least one clock cycle before changing the data again.

Read about the Coroutines and Tasks in the official documentation of *cocotb*. Some examples can also be found there.

Observe the test results and the waveforms.

- Does the waveforms look as expected?
- What happens when changing the waiting time after assigning the data?

Rerun the test for different parameters of the DUT, for example:

```
# Re-run the tests
(.venv) [<username>@<servername> sat_cocotb_exercise]$ make clean
(.venv) [<username>@<servername> sat_cocotb_exercise]$ make THRESHOLD=5
```

Part 2: Constraints

Now, update the `sat_random_test.py` test but constraining the input value using the *PyVSC* library. For example, the input data can be constraint in order to generate only positive values bellow 10.

Read more in the *PyVSC* Data Types documentation and *PyVSC* Constraints documentation.

Exercise 4: Random Virtual Sequence

Objective: Introduction to the randomization and constraint concepts using the *PyVSC* library.

Exercise: Creation of a virtual sequence and a test case to run the virtual sequence.

Create a new virtual sequence that randomizes the number of sequence items generated by the agent. Use the provided template 7.1 as the starting point to create the virtual sequence while following the next steps:

1. Inside the folder `<ROOT>/sat_filter/src/tb/vseqs/`, create a file called `sat_filter_rnd_itr_seq`, use the file `sat_filter_default_seq` as reference.
2. The new virtual sequence must be extended from the `sat_filter_tb_base_seq`.
3. The new virtual sequence must contain a random variable that shall be used to control the number of items sent to the *uVC*. This variable must be called `itr_nbr`. Read more in the *PyVSC* Data Types documentation.
4. Define a constraint for the `itr_nbr` variable in order to generate only positive values when randomized. Read more in the *PyVSC* Constraints documentation.
5. In the virtual sequence `body` two coroutines must be implemented: one for the producer agent and a second for the consumer agent of the *sSDT*. In this loop, over the `itr_nbr` value, the coroutines must be launched in parallel.

```python
@vsc.randobj
class sat_filter_rnd_itr_seq(sat_filter_base_seq):

    def __init__(self, name="sat_filter_rnd_itr_seq"):

        super().__init__(name)

        # TODO: Add missing sequences declaration
        # TODO: Add itr_nbr parameter

    async def body(self):

        # Launch sequences
        await super().body()

        # TODO: Fill in with the missing steps

    @vsc.constraint
    def itr_nbr_c(self):

        # TODO: Add itr_nbr constraint here
```

Listing 7.1: Template for the `sat_filter_rnd_itr_seq` sequence.

For the execution of the virtual sequence, a test case must be created. Use the provided template 7.2 for the following steps:

1. Create a new test case, named `test_sat_filter_rnd_itr`, inside `<ROOT>/sat_filter/src/tb/tests/`.
2. This test case must be extended from `sat_filter_base_test` provided inside `<ROOT>/sat_filter/src/tb/tests/`.
3. The sequence must be added in the test using the factory override method in the `start_of_simulation_phase`.
4. The `run_phase` must also be implemented accordingly: first randomizing the virtual sequence and then starting the virtual sequence on the virtual sequencer.
5. Improve the test case by adding an inline constraint to control the randomization from the test case.

```python
@pyuvm_test
class test_sat_filter_rnd_itr(sat_filter_base_test):

    def __init__(self, name="test_sat_filter_rnd_itr", parent=None):

        super().__init__(name, parent)

    def start_of_simulation_phase(self):

        super().start_of_simulation_phase()

        # TODO: Complete (...)
        # uvm_factory().set_type_override_by_type(...)

    async def run_phase(self):

        self.raise_objection()
        await super().run_phase()

        # Randomize sequence
        self.virt_sequence.randomize()

        # Run sequence
        await self.virt_sequence.start(self.tb_env.virtual_sequencer)

        self.drop_objection()
```

Listing 7.2: Template for the `test_sat_filter_rnd_itr` test case.

Exercise 5: Scoreboard implementation

Objective: Introduction to the UVM Scoreboard component.

Exercise: Implementation of the Scoreboard component to compare the output of the *Saturation Filter* with the Reference Model results.

The results from the DUT and the Reference Model need to be compared to ensure the correct behavior of the design. In order to compare, these results must be stored and share between the testbench components. For that purpose, queues are used.

The DUT's input is sent via the Producer Agent's analysis port to a FIFO in the Reference Model. Then, the output of the Reference Model is sent through the analysis port to a FIFO in the Scoreboard. Through the Consumer Agent, the DUT's output is then shared via the analysis port to another FIFO in the Scoreboard. In the Scoreboard the content of the FIFOs is copied to separated queues which are then used to compare the results.

A simple implementation of the Reference Model code can be found in `<ROOT>/sat_filter/src/tb/ref_model/` and the UVM class can be found in `<ROOT>/sat_filter/src/tb/sat_filter_ref_model.py`.

A skeleton of the Scoreboard class is implemented in the file, `<ROOT>/sat_filter/src/tb/sat_filter_scb`. Notice that all the required components are already implemented in the `build_phase`.

The Scoreboard implementation must be completed as follows:

1. Implementing the missing steps to get the items from the available FIFOs and compare against each other. If the items are matching, the `success` variable must be increased, if not an error message must be printed and the `failure` variable increased.
2. Implement the `check_phase` to print the value of the `success` and `failure` variables.

Now the Scoreboard must be integrated in the testbench, specifically in the `environment` component, located in `<ROOT>/sat_filter/src/tb/sat_filter_tb_env`. For that, the next steps must be followed:

1. First, the Scoreboad handler must be included in the class constructor, then instantiated in the `build_phase`.
2. Connect the Consumer Agent's analysis port with the Scoreboard consumer's FIFO, created before.
3. Connect the Reference Model's analysis port with the Scoreboard reference model's FIFO, created before.

Exercise 6: *sSDT* Protocol Checkers

Objective: Introduction to the concept of protocol checkers as parallel coroutines to the simulation.
Exercise: Implementation of protocol checkers to ensure that the *uVC* is behaving according to the protocol specifications.

The checkers should be implemented as Python coroutines, which are running in parallel with the other processes of the simulation. Follow the implementation steps as follows:

1. Locate the file `uvc_ssdt_interface_assertions`, inside the folder `<ROOT>/sat_filter/src/tb/uvc/ssdt/src/`.
2. Add the protocol checkers inside this file. e.g.: a coroutine should check that the `data` is never `X` when the `valid` signal is high.
3. Open the file `<ROOT>/sat_filter/src/tb/sat_filter_tb_base_test.py`, and:
   - Connect the DUT input signals (`clk, rst, in_data, in_valid`) to the `ssdt_interface_assert_check()` class.
   - Start the checking coroutine in the `run_phase` of the test.
4. Repeat the same process for the DUT output signals.

Exercise 7: Reactive Test

Objective: Overview of the connection between the several UVM components.

Exercise: Develop a test case that reacts to the sequence response item of the consumer agent and stop the simulation if a certain condition is met.

Create a new test case which shall be named `test_sat_accu`, inside `<ROOT>/sat_filter/src/tb/tests/`, by following the next steps:

1. Create a test case extended from the `sat_filter_tb_base_test`.
2. Create a new variable, called `accumulator_max`, that will set the maximum value of the accumulated response. Set it to an arbitrary value, e.g. 200.
3. Create a new variable, called `accumulator_value`, that will store the accumulated value.
4. In the test, reuse the default sequence that generates a single item to the *sSDT uVC*.
5. An interruption mechanism must be implemented as follows:

   After the sequence is finished the test should increment the variable `accumulator_value` with the value of the data from the response item of the consumer agent. Based on the response, perform one of the followings:

   - If `accumulator_value` is less than the `accumulator_max`, the loop must continue.
   - If `accumulator_value` is equal or higher than the `accumulator_max`, the loop must be interrupted.

Exercise 8: *PyVSC* Coverage in *cocotb* test

Objective: Introduction to coverage collection using the *PyVSC* library.

Exercise: Implementation of a coverage collector in the *cocotb* test exercise using the *PyVSC* library.

Locate the *cocotb* exercise in `<ROOT>/exercises/sat_cocotb_exercise/`. Create a file to implement the *coverage collection* with the name `sat_filter_coverage.py`. Inside the file, create a `covergroup` called `covergroup_ssdt`. The `covergroup` must contain a `coverpoint` for the `data`. Read more in the *PyVSC* Coverage documentation.

Now, the coverage collector must be introduced in a test. Locate the random test, implemented using the constraints, created in "Exercise 3: *cocotb* Test" and integrate the coverage collector created. The coverage collector must sample the `out_data` signal when `out_valid` goes high.

The test should generate a coverage report before ending. Use the provided method `create_coverage_report` from `utilities.py`. The `create_coverage_report` requires as input the name of the test case, e.g., `create_coverage_report("sat_random_test")`.

Look at the coverage results for each test inside `sim_build`. Analyze the files `<test-name>_cov.txt` generated. The `PyUCIS-viewer` tool can also be used to visualize the coverage results, by running, e.g.:

```
(.venv) [<username>@<servername> sim_build]$ pyucis-viewer sat_random_test_cov.xml
```

Analyze the coverage report. How good are the results for the coverage using the test? Try to change the `bins` in the `coverpoint` and see how does that affect the coverage results.

Exercise 9: *PyVSC* Coverage for *sSDT uVC*

Objective: Introduction to the coverage collection in a UVM testbench using the *PyVSC* library.
Exercise: Implementation of the coverage collector class in the *sSDT*'s *uVC*.

Locate the coverage collector file, `uvc_ssdt_coverage.py`, inside `<ROOT>/sat_filter/src/tb/uvc/ssdt/`. The file contains a partial implementation of the class `uvc_ssdt_coverage`. The implementation must be completed respecting the following specifications:

Create the `covergroup` class, named `covergroup_ssdt`, with the following requirements:

- Must define a `coverpoint` for the `data` field of the *sSDT* sequence item.
- The `coverpoint` must cover the following bins:
  - `data` when is 0;
  - `data` when is the maximum value;
  - `data` in the range between 0 and the maximum value.

The `covergroup` should be integrated in the `uvc_ssdt_coverage` class, as follows:

- Correct instantiation in the `build_phase`;
- Implementation of a `write` method inside the `uvc_ssdt_coverage` class, as follows:
  - Must take an item (`uvc_ssdt_seq_item`) as input parameter;
  - Must call the `sample` method of the `covergroup` by passing the item, e.g., `self.cg_ssdt.sample(item.data)`.

The coverage collector should be integrated in the agent, located in `<ROOT>/sat_filter/src/tb/uvc/ssdt/src/uvc_ssdt_agent.py`, as follows:

- Integrate the handle for the `uvc_ssdt_coverage` in constructor of `uvc_ssdt_agent` class;
- Create an instance for the `uvc_ssdt_coverage` in the `build_phase`;
- Share the handler in the `ConfigDB`;
- Connect the monitor's analysis port to the coverage's analysis export in the `connect_phase`.

Analyze the coverage report files (*.xml) generated inside `<ROOT>/sat_filter/src/tb/sim_build` by running the command:

```
(.venv) [<username>@<servername> tb]$ pyucis-viewer \
sim_build/test_sat_filter_default_seq.xml
```

Exercise 10: *PyVSC* Coverage for *Saturation Filter* testbench

Objective: Introduction to the coverage collection in a UVM testbench using the *PyVSC* library.
Exercise: Implementation of the coverage collector class for the *Saturation Filter* testbench.

Develop the coverage collector class for the *Saturation Filter* testbench by creating a new file, named `sat_filter_coverage.py`, inside `<ROOT>/sat_filter/src/tb`. The coverage collector class must be named `sat_filter_coverage` and extended from `uvm_subscriber`.

Following the same steps to implement `uvc_ssdt_coverage.py` in exercise "Exercise 9: *PyVSC* Coverage for *sSDT uVC*", implement a `covergroup` class to cover the `ovf` signal from the output of the DUT.

Notice that the `ovf` signal is not being monitored by any available `monitor` component. Because of this, a coroutine must be implemented.

In the `sat_filter_tb_base_test` create a coroutine that will run concurrently to the test. This coroutine must call the `write` method of the `sat_filter_coverage` and pass the `ovf` value whenever `out_valid == 1`. This coroutine must be launched in the `run_phase`. Remember also to create the `sat_filter_coverage` in the `build_phase` of the `sat_filter_tb_base_test`.

Run the test and look at the coverage results inside `<ROOT>/sat_filter/src/tb/sim_build`. Use the `PyUCIS-viewer` tool to visualize the coverage results.

Exercise 11: Coverage Holes Analysis

Objective: Overview of the concepts of coverage reports and coverage holes.

Exercise: Generation of coverage reports for different parameters of the DUT.

Analyze the coverage results obtained from the previous exercises. Is there something that can be done to increase the percentage covered?

What happens if the `THRESHOLD` parameter is increased and the `DATA_W` parameter decreased?

Rerun the tests for different combinations of the DUT's parameters and analyze the results using the `PyUCIS-viewer` tool.

After some tests merge the coverage results and visualize it by running:

```
# The following command will:
# 1. merge all available "*.xml" files inside "sim_build/" and creates a "merge_cov.xml"
file.
# 2. open pyucis-viewer with the "merge_cov.xml" file

(.venv) [<username>@<servername> tb]$ make coverage
```