UVVM Tutorial

A brief introduction to UVVM, with a
tutorial on how to use it in an example testbench.

University of Applied Sciences Upper Austria
Campus Hagenberg

Embedded Systems Design

Practical FPGA Applications (PFA2)

Michael Wurm

2nd Semester, 2020

April 18, 2020

Contents

FH-OOE / Hagenberg / ESD

Michael Wurm © 2020
Practical FPGA Applications
Introduction and tutorial to UVVM

1 Prerequisites

This tutorial requires the ModelSim simulator software to be installed. Furthermore, it is highly recommended to work on a PC that runs a Linux operating system. The tutorial was developed using Ubuntu 18.04 LTS.

2 Preparation

This tutorial aims to explain UVVM and work with it. Thus, we do not want to lose time with the environment setup. So in order to be able to follow the tutorial, please complete the following steps in advance.

2.1 Clone the tutorials' repository

Several sources required by this tutorial can be found in a GitHub repository.
Download, or clone the repository.

```
git clone --recursive https://github.com/wurmmi/uvvm-tutorial.git
```

2.2 Install ModelSim

Any version of ModelSim should satisfy this tutorials' requirement.

In case you are using ModelSim Intel Starter Version, you might need to create a symlink in the installation directory.

- Go to the ModelSim installation directory
  (e.g. `/opt/intelFPGA/19.1/modelsim_ase`)
- Create a symlink from `linuxaloem` to `linuxpe`.
  (`sudo ln -s linuxaloem/ linuxpe`)

Verify, that ModelSim can be found. Execute the following command in a command line.

```
vsim -version
```

In case this fails, please work through the section "Install Intel Modelsim" in this installation guide (doc/install-modelsim-on-ubuntu.pdf).

3 UVVM Overview

UVVM, short for Universal VHDL Verification Methodology, is a VHDL testbench in-frastructure, architecture, library and methodology. It is developed by Bitvis, acompany with its headquarter located in Norway.

The library is maintained as an open-source project, where developers can contributeto the existing code base.

3.1 General

UVVM consists of two major parts, which are the Utility Library and the VVC Framework.

The Utility Library provides basic fundamental features, such as logging, alert handling and results checking.
The VVC Framework builds an abstraction layer around the Utility Library and adds extended functionality on top of it. The major functionality there is the capability of scheduling and skewing actions with respect to each other. This means, stimuli have the ability to wait for others to finish, before they start. This especially simplifies verification in a testbench.

3.2 Advantage Over Other Verification Libraries

Different libraries and frameworks can be used for verification of an RTL design. They make use of different programming languages, such as SystemC or SystemVerilog. The masters courses *Advanced Methods of Verification (AMV2)*, or *Hardware-Software Co-Design (HSC2)* talk about these in detail.

In order to use those languages, simulators need to be capable to work with those languages. This often requires to buy a license for language support. As a company, license costs are a factor to consider. Cost optimization in development processes is constantly driven forward. Thus, avoiding the need of a simulator license with support for several languages can help lowering costs.
UVVM uses VHDL only, which - for above mentioned arguments - is a good reason to choose it for a project. Another reason is, that it is an open-source library and is continuously developing and improving.

3.3 Structure and Architecture

The UVVM testbench architecture offers a well-structured design that can be extended in a modular way. Readability is maintained by having global signals, that are hidden inside the library. In that way, even complex testbenches for large designs can be kept readable, as the amount of interface signals can be kept low. Following the modular structure, each interface is attached to its own verification component. Thus, the amount required to adapt a testbench to a changed interface on the DUT is minimal.

Fig. 1 shows an example testbench architecture. It includes the test sequencer, the test harness and the DUT. Furthermore, it shows that each interface is attached to its' own VHDL verification component (VVC). Signals that need to be connected between those parts are displayed in solid lines, signals that are handled by UVVM in the background in dashed lines.

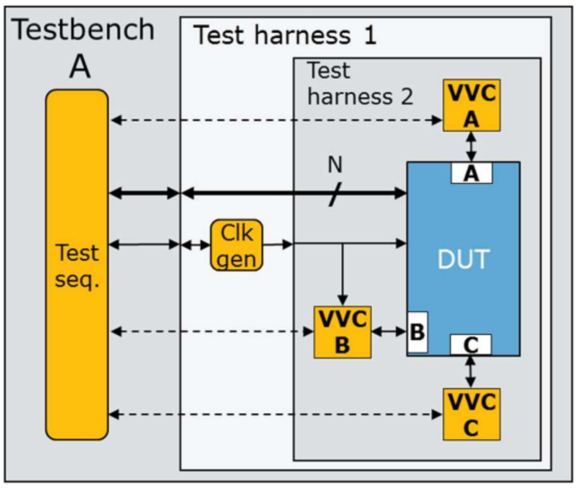

Figure 1: *Testbench architecture*

The modular architecture reveals its advantages especially when the DUT is changing its interfaces, as already mentioned above. For example, a DUT has an AXI memory mapped bus interface that the testbench is exercising extensively and in several locations in the testbenches' code. For some reason, the designer now replaces it with an Avalon memory mapped bus interface.
For the UVVM testbench this only means to swap the AXI VVC with an Avalon VVC. All remaining functionality can be kept as it is. Only minimal adaptions are required, instead of re-writing the whole testbench part, that handles the changed interface.

3.3.1 Testharness

The test harness contains the instantiation of the actual DUT. Furthermore, the VVCs to handle all interfaces are instantiated here.

3.3.2 Testbench

The actual testing procedure happens in the testbench. It includes the main test sequencer, where all the test procedures are exercised.

To maintain readability in complex designs, test procedures can be implemented in separate VHDL packages.

3.3.3 VHDL Verification Component

A VHDL Verification Component, short VVC, provides methods to verify a specific type of interface. Those methods are high-level abstractions of the transactions that this interface can perform (e.g. *read* and *write*).

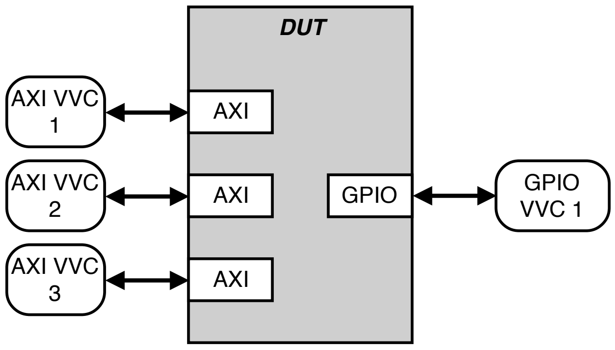

Figure 2: *VVCs connected with a DUT*

Examples of VVC functions are `axi_write(...)` or `gpio_expect(...)`. Those functions internally call methods of their respective bus functional models (BFM) that handle the low level actions.

3.3.4 Bus Functional Model

A Bus Functional Model, mostly referred to as BFM, handles the low-level interactions for a respective interface. The actual signal transactions for an interface are set here, according to the specific protocol. This includes handshake signals, such as read-valid, write-valid, various ready signals, wait states, busy signals and the actual data transmission lines.

3.3.5 Verification IP

A Verification IP, short VIP, basically acts as a container in the UVVM directory structure. It summarizes VVC and BFM, as well as corresponding documentation. Scripts to compile the VIP accordingly are provided as well.

4 Tutorial

This tutorial guides you through the example project that was developed.

4.1 Device Under Verification (DUV)

The Device Under Verification (DUV) is a very basic LED driver, that has some additional accessible memory inside. The interface this memory uses, is compile-time configurable between an AXIlite, or an Avalon MM protocol.

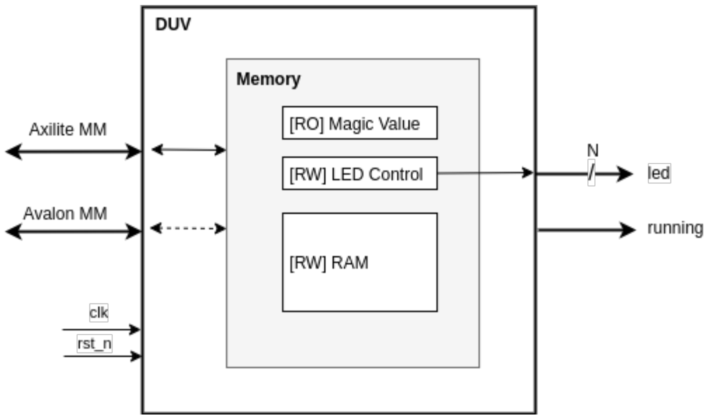

Figure 3: *DUV Block Diagram*

4.2 Testbench Architecture

The architecture of the demo testbench is shown in the block diagram below.

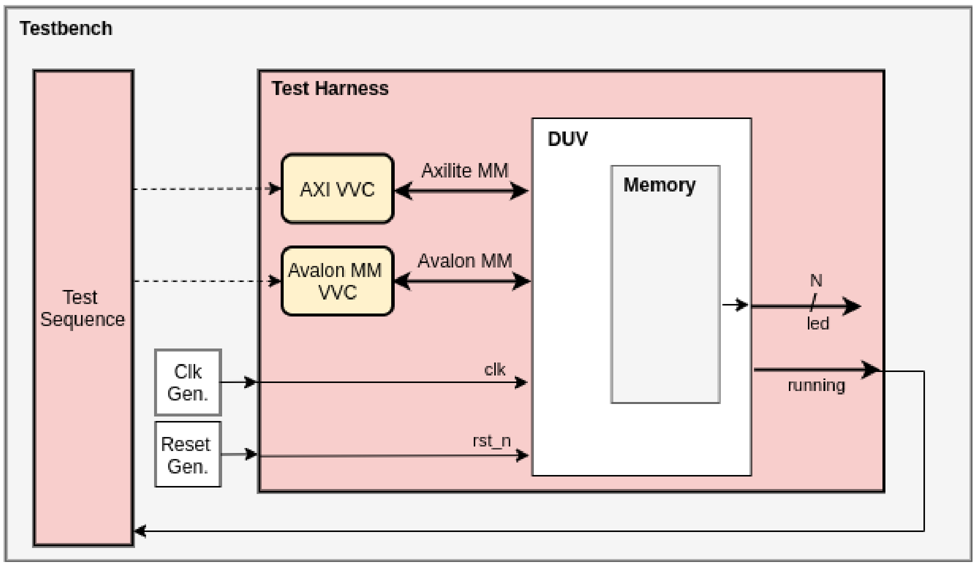

Figure 4: *Testbench Block Diagram*

The testbench instantiates a test harness and runs the actual test sequences. It further generates clock and reset signals for the DUV. The test harness instantiates several VVCs and the DUV. Dotted lines in the block diagram represent connections, that exist within the UVVM framework and don't need to be connected explicitly.

4.3 Hands-On

At this point, we have enough information and are ready to start working with the example project.

4.3.1 Project environment

Let's start with getting to know the projects' environment setup.
In this part we will compile the UVVM library and the DUV, and run the simulation of our first example testbench!

1. Open the example-tb folder in the repository. You will find a Makefile, that's used to simplify the usage of this project. It builds an abstraction layer to the underlaying ModelSim commands.

2. `make help` is your friend.
   It shows all the actions, that this project provides.

3. Call `make ip`.
   This compiles all the VHDL files, that belong to the DUV.
   Call this target, whenever your DUVs RTL code changed and you want to simulate it.

4. Call `make uvvm`.
   This compiles several required components of the UVVM library.
   That only needs to be done once.

5. Call `make test`.
   This compiles the testbench files and runs the simulation.

6. Read through the log that was generated.
   The console output was also logged into a log file[1].

4.3.2 Read through the AXI test sequence

The DUV includes a memory that can be accessed through an AXI memory mapped interface. In this part, we will read through the existing AXI test sequence code.

1. Open the AXI register test sequence file[2].

2. Check out the main AXI VVC functions.

   - `axilite_write()`
     Writes data to a given address.
     This function is non-blocking. It schedules the action in the UVVM sequencer queue.
   - `axilite_check()`
     Reads data from a given address and compares with the given value.
     This function is non-blocking. It schedules the action in the UVVM sequencer queue.
   - `await_completion()`
     This function waits on a given VVC to complete its action queue. In case the VVC exceeds the given timeout, an error is raised.

[1]example-tb/build/log/sim_testbench.log
[2]example-tb/src/tb/sequences/axi_reg_seq_pkg.vhd

3. Try to write a register that is read-only.

   - Writing to a read-only register is not allowed. Thus, we expect the call `axilite_write()` to fail and raise a `TB_FAILURE`.
   - We need to tell UVVM that we are expecting this failure. By default, the testbench execution is stopped after the occurance of a single failure.
   - Two functions need to be called to set expected failures.
     These are `increment_expected_alerts()` and `set_alert_stop_limit()`.
   - At the end of the testbench execution log, there is a report in the form of a table, that shows the number of alerts that occurred during the simulation run. See, that the number of *expected* alerts matches the number of *regarded* alerts. (The latter should better be called *occured* in my opinion...).
     However, this means that writing to the read-only register failed as expected, and our test case was successful.

4.3.3 Write a memory test sequence

In this part, we will write a test sequence that verifies the correct functionality of this memory.

1. Open the AXI register test sequence file[3].

2. Scroll to the end of the file, where you will find a "TODO" section.

3. Write a test sequence, that runs a write-read test on all registers of the internal memory.

   For information about the memory structure, look at the C include file[4].
   Furthermore, taking a look at the already existing test sequence for the Avalon MM interface is a good point to begin with. Find it in the Avalon MM test sequence file[5].

4. Run the test sequence with `make test`.
   Remember, that this requires UVVM to be built once (`make uvvm`) and the DUV to be built (`make ip`) each time you changed the DUVs' RTL.

5. Your test sequence should run successfully and output logs to the command line. Look at these logs, to evaluate your test sequence.

4.3.4 Realize the similarity between the different interfaces

At this point, we have two working test sequences for the memory. One that uses an AXI interface, and another that uses an Avalon MM interface.

By comparing the two test sequences, we realize, that they are pretty much the exact same code. This is a major advantage of using VVCs. Imagine the following situation. We have a working DUV, that has an AXI interface. For some reason, the designer decides to swap it against an Avalon MM bus. In a simple "selfmade" testbench, this would require a major code change. However, in our UVVM verification environment, this simply means to switch out the VVC component, re-wire the bus signals in a single place in code, and rename the VVC functions (e.g. `axilite_write()` to `avalon_mm_write()`).

[3]example-tb/src/tb/sequences/axi_reg_seq_pkg.vhd
[4]example-tb/src/include/blinkylight.h
[5]example-tb/src/tb/sequences/av_mm_reg_seq_pkg.vhd

4.3.5 (optional) The scoreboard

The UVVM scoreboard is a useful tool to acquire some statistical data with test sequences.

1. Open the Avalon MM test sequence file[6].
2. Scroll to the write-read test section.
3. Familiarize with the scoreboard functions that are used.
4. Run `make test` and find the log output that was produced by the `scoreboard.report_counters()` function.

4.3.6 (optional) Write a LED test sequence

The DUV acts as a driver to an array of eight LEDs. The respective value the LEDs display are a representation of the `LED_CONTROL` register.

To verify the correct functionality of this LED driver part, we will implement an extension to the AXI test sequence we previously worked on.

1. Open the test harness file[7].
2. Scroll to the "instantiations" section of the file, where you will find a GPIO VVC that is already instantiated and hooked up to the LED output of the DUV.
3. Open the AXI register test sequence file[8], which we already know from a previous task in the tutorial.
4. In the existing test sequence, there is a part that sets a value to the `LED_CONTROL` register and reads back its value, to verify the register access. However, this does not verify that the value was actually routed through all the way to the LED output. Let's verify that.
5. Use the GPIO VVC to check the value of the LED output.

   Each UVVM VVC is provided along with documentation. Open the documentation for the GPIO VVC[9] to see which functions are available.

   A little hint:
   Use the `gpio_expect(VVCT, vvc_instance_idx, data_exp, timeout, msg)` function.
6. Run `make test` to run the test sequence.

[6]example-tb/src/tb/sequences/av_mm_reg_seq_pkg.vhd
[7]example-tb/src/tb/blinkylight_th.vhd
[8]example-tb/src/tb/sequences/axi_reg_seq_pkg.vhd
[9]UVVM/bitvis_vip_gpio/doc/gpio_vvc_QuickRef.pdf

4.4 Solutions to the implementation parts

In this section, the solution to several implementation parts of this tutorial is revealed.

1. Open the repository in a command line, or your favourite Git GUI tool (e.g. GitKraken).
2. Checkout the branch called `tutorial-solution`.
   In the command line run `git checkout tutorial-solution`.
3. Now navigate to the respective files that required some implementation and see the solution!