UVVM VVC Framework

Manual

Rev. A6

Document Change History

Revision	Date	Change
A1	2015.12.07	Documentation for first public release (v1.0.0) of VVC Framework
A2	2016.01.20	Updated for open source release (v1.0.1) of VVC Framework. Removed section 1.4 Encrypted Code since the code is open source now.
A3	2016.11.08	Updated sections 1.1.2 and 1.1.3 with libraries and install procedure.
A4	2017.02.09	Updated with full implementation of Avalon MM, multiple sequencers.
A5	2017.09.29	Updated Acronym/Abbrevation table, section 1.2, Table 2. Figure 6, Figure 7.
A6	2022.09-10	Removed links to other methodologies now covered by UVVM.

Acronyms and Abbreviations

Acronym / Abbreviation	Definition
Avalon-MM	Avalon Memory Mapped
BFM	Bus Functional Model - Basically a set of procedures that as a total can mimic the behaviour of a physical interface like a UART or AXI4-Lite. - A BFM for a UART would typically consist of minimum two procedures: Transmit() and Receive(). Additional procedures could be added, like Expect(), which would typically call Receive() and check the received data against the expected data. - BFMs here are only meant to feed or get data in/out of a physical interface via a given legal variant of the required protocol. Hence protocol checking is not included unless otherwise noted.
CDM	Command Distribution Method In VVC FRAMEWORK this is almost the same as a BFM, with one single difference. The BFM is executed towards the physical DUT immediately, whereas a CDM is always used to **only distribute** executable commands to a VVC for execution there (often via BFM inside the VVC) – immediately or later. Hence a CDM is never wiggling signals on a physical DUT.
DUT	Device Under Test (meaning Verification in this case)
GPIO	General-purpose Input/Output
I2C	Inter-Integrated Circuit
PIF	Processor InterFace
SBI	Simple Bus Interface. A single cycle bus interface as simple as can be, using CS, ADDR, RD, WR,

	RD_DATA, WR_DATA and optional READY.
SPI	Serial Peripheral Interface
TB	TestBench
UART	Universal Asynchronous Receiver/Transmitter
UVVM	Universal VHDL Verification Methodology
VIP	Verification IP. Used as a common notation for all types of verification IP. Often includes a BFM and VVC. May also include additional verification IP.
VVC	VHDL Verification Component

About UVVM VVC Framework

UVVM VVC Framework is an optional part of UVVM (Universal VHDL Verification Methodology) and provides support to implement a very structured testbench architecture. This architecture allows significant efficiency improvement for the verification of modules or FPGAs with two or more interfaces, where these interfaces need to be controlled or monitored simultaneously – typically in order to reach corner cases inside the DUT (Device Under Test) – or just to get a better overview and control over your stimuli, checkers and monitors.

VVC Framework will be used as a short form for UVVM VVC Framework.
Similarly Utility Library (or sometimes UVVM Util) will be used as a short form for UVVM Utility Library.

VVC Framework was originally intended as a system and methodology to detect cycle related corner cases by allowing skewing of actions on one interface with respect to another in an easily understandable manner. This resulted in the testbench architecture and mechanisms needed to support the very structured simultaneous control of stimuli and checks on multiple interfaces.

The significantly improved testbench overview, maintainability, extendibility and reuse friendliness of this system has however also proved to be extremely valuable to detect most other types of corner cases. Thus VVC FRAMEWORK is an excellent platform for verifying any complex VHDL based module or FPGA.

Please see the attached PowerPoint '*The_critically_missing_VHDL_TB_feature.ppsx'* for a presentation on cycle related corner cases and the need for a far more structured verification approach.

Prior to VVC FRAMEWORK, verification of delta cycle related corner cases was handled as follows:

- In many cases not handled at all, but ignored due either to lack of knowledge or ignoring the problem - just hoping or assuming that the design is correct by construction
- Hoping or assuming the corner cases will be detected in the lab
- Making an ad-hoc, unstructured testbench

- Making a relatively structured, but far too complex testbench over which it is really difficult to get a good overview

There were in fact no good solutions that provided a good structure, a good sequencer-VVC communication, a good overview and a good methodology.

The consequences of this have been:

- Inefficient testbench development, extensions, modifications
- Difficult to reuse TB parts in a project - or to share the TB itself
- High risk of missing critical corner cases

With VVC FRAMEWORK this has changed and we can now achieve:

- Major time saving (several man-weeks or man-months for medium complexity FPGAs)
- Significant quality improvement for end product
- A new world for overview, maintainability, extendibility and reuse

The PowerPoint presentation referenced above is assumed read before going further in this manual.

Please note that VVC Framework is using UVVM Utility Library (UVVM Util) as a basic testbench infrastructure with support for logging, alert handling, verbosity control, checkers, awaits, etc. UVVM Util is provided with the full VVC FRAMEWORK download, but may also be downloaded separately – to make it simpler for users who do not need the full VVC FRAMEWORK.

UVVM License and Disclaimer may be found in section 5

Table of Contents

1 Quick Start Guide

This Quick Start Guide will briefly guide you through the installation process. For detailed technical reference see section 2.

1.1 Installation

1.1.1 System Requirements

There are no system requirements other than a VHDL 2008 compatible simulator.

Note:
UVVM requires a VHDL 2008 compatible simulator. Currently only the simulators from Aldec and Mentor Graphics have sufficient VHDL 2008 support.

UVVM has been tested with the following simulators:

- Modelsim version 10.3d
- Riviera-PRO version: 2015.10.85

It should be notes that Python 3 is required if you want to execute the vvc_generator or vvc_name_modifier script to make new VVCs in a simple way.

1.1.2 Bundled Libraries

VVC FRAMEWORK is bundled with libraries as listed in Table 1.

Table 1 Libraries bundled with VVC FRAMEWORK.

Library	Description	Location
UVVM Utility Library	UVVM Utility Library is an open source VHDL test bench (TB) infrastructure library for verification of FPGA and ASIC. Used by VVC FRAMEWORK as a common testbench infrastructure.	<install_dir>/uvvm_util
UVVM VVC Framework	The library for the VVC Framework with the functionality described in this document.	<install_dir>/uvvm_vvc_framework
bitvis_vip_sbi	VIP including a BFM and VVC for a simple bus interface (SBI). This VVC is intended as a template for writing new VVCs and for understanding the VVC functionality. This library is also used in the provided testbench example.	<install_dir>/bitvis_vip_sbi

bitvis_vip_uart	VIP including a BFM and VVC for a simple UART interface. This VVC may be used as a template for writing new VVCs for multi-channel interfaces. This library is also used in the provided testbench example.	<install_dir>/bitvis_vip_uart
bitvis_uart	This is a simple UART design that is being used as a DUT for the provided example testbench	<install_dir>/bitvis_uart
bitvis_vip_axilite	VIP including BFM and VVC. This simple AXI4-Lite BFM and VVC is provided as a kick start for users to make their own testbenches using VVC FRAMEWORK, as many designs today have an AXI4-lite interface.	<install_dir>/bitvis_vip_axilite
bitvis_vip_avalon_mm	VIP including a BFM and VVC for an Avalon-MM interface.	<install_dir>/bitvis_vip_avalon_mm
bitvis_vip_axistream	VIP including a BFM and VVC for a simple AXI-Stream interface.	<install_dir>/bitvis_vip_axistream
bitvis_vip_i2c	VIP including a BFM and VVC for a simple I2C interface.	<install_dir>/bitvis_vip_i2c
bitvis_vip_spi	VIP including a BFM and VVC for a simple SPI interface.	<install_dir>/bitvis_vip_spi
bitvis_vip_gpio	VIP including a BFM and a VVC for a simple GPIO interface.	<install_dir>/bitvis_vip_gpio

1.1.3 Installing and compiling VVC FRAMEWORK

1. Download the UVVM package by cloning the UVVM repository from GitHub: https://github.com/UVVM/UVVM_All
2. If UVVM was downloaded as a zip file, extract the downloaded zip-file to a directory of your choice, making sure that all the directories for the various parts of VVC FRAMEWORK, VVCs and Testbench are located as given in the table above
3. Compile all files as given in the respective QuickReferences for all parts of the VVC FRAMEWORK and VVCs (*1)

If you want to run the provided testbench also do the following:

4. Compile the DUT and TB for the UART as given in the compile scripts there (*1)
5. Elaborate and Run the testbench for the UART. (*1)

*1: For Modelsim users all compilation, elaboration and running the simulation could be handled automatically by running the provided scripts in the various directories. In the script catalogue of the UART there are hierarchical scripts to run all necessary scripts. If you import the .mpf-file in the UART sim-directory the script files will be shown inside

the Modelsim project environment, and all you have to do is to right click the scripts and execute them.

1.2 Help and bug reporting

For help, please read the provided documentation and have a look at the UART example testbench in 'bitvis_uart/tb/uart_vvc_tb.vhd'.

For bug report, please create an issue on GitHub - https://github.com/UVVM

1.3 Patents

There are patent issues pending for several parts of VVC FRAMEWORK.
These patents are only intended to avoid theft of the complete UVVM or critical concepts. They are not in any way restricting the use or modification of UVVM – other than what is already defined in the license.

2 Understanding the VVC Framework

Anyone using this system should read this part prior to making any modifications to the provided examples and definitely before making your own VVC FRAMEWORK based testbench.

2.1 Prerequisites

It is strongly recommended before you commence that:

1. You have gone through the introduction PowerPoint presentation for VVC FRAMEWORK
   See '*The_critically_missing_VHDL_TB_feature.ppsx*'

2.2 Understanding the VVC FRAMEWORK testbench architecture

The VVC FRAMEWORK testbench architecture is simple to understand (see the two different testbench approaches in Figure 1.

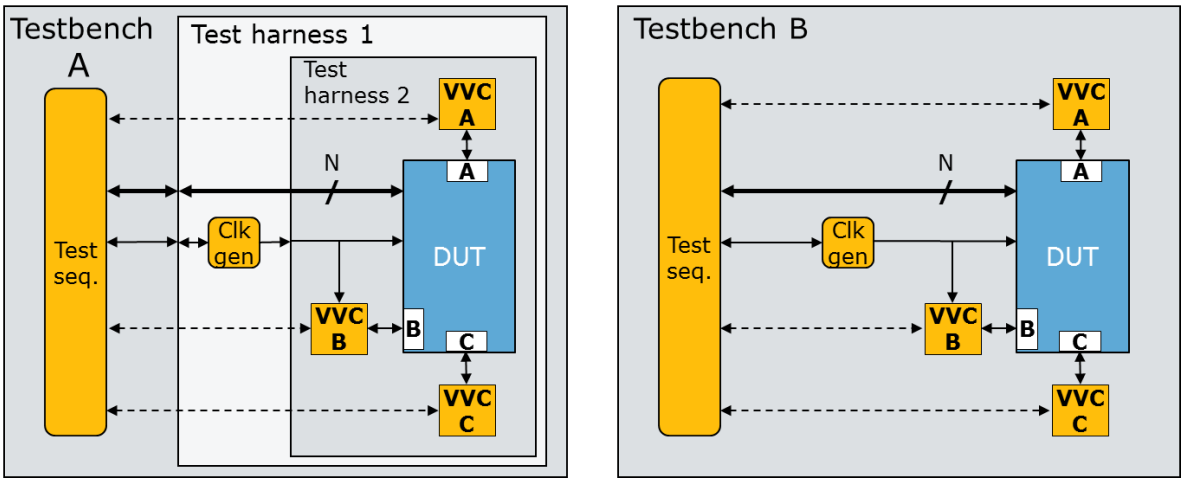

Figure 1: Testbench architecture

In both testbenches the interfaces on the DUT (here A, B, C) are connected to the corresponding verification components (VVCs A, B, C) as any other inter entity connections. Any support process like for instance a clock generator is connected to the DUT as normal. The clock generator could be totally independent - or controlled from the sequencer as indicated here.

The figure shows a DUT with three different interfaces. In lots of systems the DUT may have several instances of the same interface, e.g. interface B. In this case two dedicated instances of VVC B (VVC_B, instance 1 and 2) must be connected to the two DUT B interfaces (B, instance 1 and 2). To differentiate between different instances of the same VVC an "Instance index" is applied as a generic input to the VVC, such that in this case one would be instance 1 and the other instance 2.

Note that no explicit connection is needed from the sequencer to the VVCs down the hierarchy – as these connections are global. They are shown as dotted lines in the figure.

2.2.1 Test harness and hierarchy

In testbench A, a test harness hierarchy is implemented to include the complete verification environment other than the sequencer. Testbench A indicates that you may indeed apply any hierarchy you want, or you can skip it all together as shown in testbench B. The global connections between the sequencer and the VVCs allow any hierarchy to be very easily added or removed. The ideal testbench would be one where all DUT interfaces are controlled via VVCs. In such a testbench there would be no signal between the hierarchical levels in the testbench, and the only signals needed would be the ones connecting the VVCs to the DUT.

The test sequencer communicates with the VVCs via global connections defined in VHDL packages. This will be explained in section 3. At this moment it is important to understand that via this communication the sequencer may distribute various commands to any VVC, and that there is a command queue inside all VVCs. These queues allow the sequencer to distribute lots of commands at the same time to the same VVC, and the commands will be executed by the VVC in the order they have been received from the sequencer; one following the other, immediately after the previous command has been executed.

There may also be multiple test sequencers – accessing different VVCs or even the same VVC.

2.2.2 VVC FRAMEWORK initialisation process

The instantiation of 'uvvm_vvc_framework.ti_uvvm_engine' is required to assure that the initialisation of the complete system is handled properly. This affects the VVC initialisation and handshake setup, and also assures that the different parts of VVC FRAMEWORK are synchronized at the start.

2.3 Understanding the VVC FRAMEWORK test sequencer

In a really simple testbench the central test sequencer will handle all the DUT interfaces directly. This would be like testbench B in Figure **1**, but without the VVCs. Hence the indicated N signals would also connect to DUT interfaces A, B and C directly.

2.3.1 Command Distribution Methods (CDM) vs BFM

Hopefully even a simple testbench will be using BFMs to access the interfaces, - as any other approach would be extremely inefficient. A simple BFM procedure call for writing to a register inside the DUT via a bus interface could typically look like the code in Figure **2**.

```
sbi_write(C_ADDR_BAUDRATE, C_BAUDRATE_10M); -- E.g. C_ADDR_BAUDRATE= x"1A",C_BAUDRATE_10M= x"15"
```

Figure 2: BFM procedure for writing to a register

This procedure when called from the sequencer will wiggle the signals of the bus interface on the DUT such that the data C_BAUDRATE is written into the register at address C_ADDR_BAUDRATE. It should be noted that while this BFM procedure is

executing, the sequencer cannot do anything - as it must now just wait until the BFM procedure is finished.

To do exactly the same using a VVC FRAMEWORK based testbench with VVCs, almost exactly the same command may be called from the sequencer, as shown in Figure 3. The only difference is the additional first parameters; - the 'target' for the command – consisting of a signal triggering the VVC and an instance number. This target specifies to where (which VVC) the command is to be distributed. The trigger signal (here 'SBI_VVCT' is given the VVC name (here 'SBI_VVC') extended by 'T' for 'Target'

```
sbi_write(SBI_VVCT,1, C_ADDR_BAUDRATE, C_BAUDRATE_10M);
```

Figure 3: CDM for writing to a register

In VVC FRAMEWORK we call this procedure a 'CDM' (Command Distribution Method) just to differentiate it from a BFM procedure. The CDMs are also called 'sequencer methods'.

The result of this method will be exactly the same as for the BFM and executed at exactly the same time towards the DUT, - because all this method does is to request the VVC (SBI_VVC) to execute the corresponding BFM procedure towards the DUT.

It should be noted that all examples of BFMs and CDMs from UVVM are slightly more advanced than the minimum BFM/CDM examples above. Our procedures have a mandatory message parameter that is used both as a comment in the sequencer code and as a transcript/log as a progress report. Our BFM/CDMs also have built in synchronization, logging, verbosity control, etc, but the implementation complexity of these features is hidden for the users and just provide more flexibility and higher value.

2.3.2 Target VVC for CDMs

As shown above for the sbi_write() CDM the target for this method is instance number 1 of SBI_VVC. I.e. the command sbi_write() with the given parameters will be distributed to SBI_VVC instance 1. The instance number of the VVC is set as a generic parameter on the VVC when instantiating it in the test harness.

Please note that some VVCs like for instance the UART has multiple channels (Rx and Tx) that operate independently. This means that a separate interpreter, queue and executor is needed for each channel, hence basically these channels need totally separate VVCs as illustrated in Figure 4. These channels however, are almost always used as a set of receiver and transmitter, and thus it makes sense to wrap the two VVCs into a single UART VVC as shown in Figure 5. Luckily from a testbench and test sequencer point of view there is no difference – as the harness can be changed as you wish and the sequencer is still connected to the leaf VVCs via the global signals.

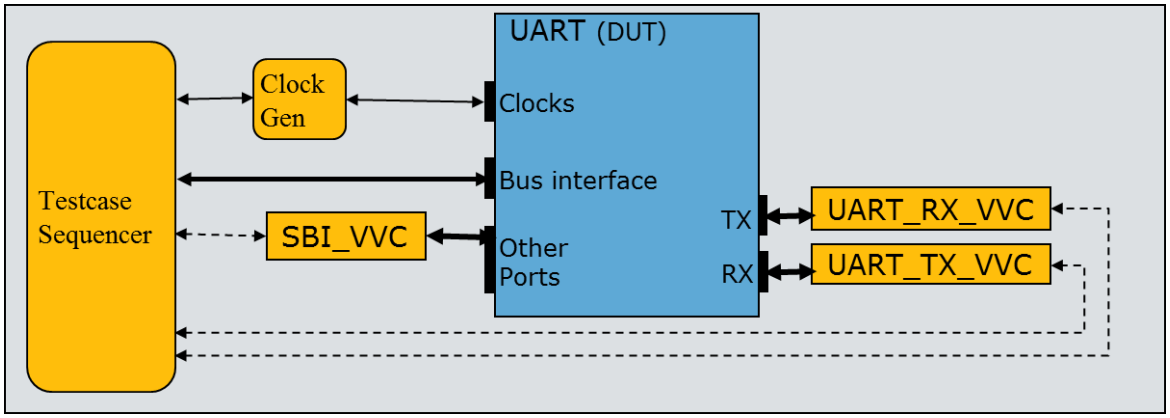

Figure 4: UART testbench using separate RX and TX VVCs

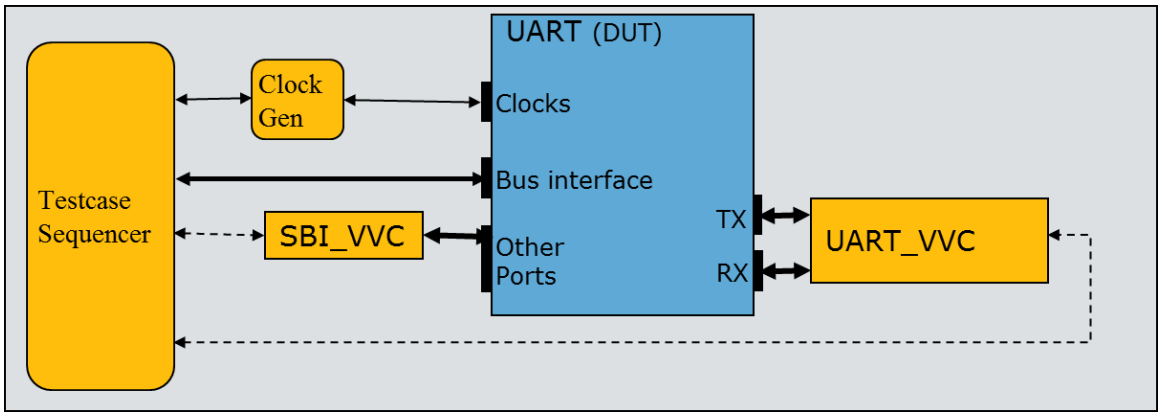

Figure 5: UART testbench using a single UART VVC

In order to support this clustering of "leaf-VVCs" into a "super-VVC", VVC FRAMEWORK allows an optional extension of the target "address" to also include the channel name.

This means that SBI_VVC, which has no channels, has a target address of VVC target signal + instance number (e.g. 'SBI_VVCT, 1') , whereas UART_VVC has a target address of VVC target signal + instance number + channel name (e.g. 'UART_VVCT, 1, RX', see Figure 7). Please note though that a VVC implementer has the freedom to use the channel specification as shown or set the target address as e.g. 'UART_RX_VVCT, 1'. There is no limitation on this in VVC FRAMEWORK.

Example target variants in VVC FRAMEWORK are shown in Figure 6.

1.	SBI_VVCT, 1	Instance number 1 of SBI_VVC
2.	UART_VVCT, 4, TX	Instance number 1 of SBI_VVC UART_VVC
3.	UART_VVCT, 3, ALL_CHANNELS	Both channels on Instance number 3 of UART_VVC
4.	SBI_VVCT, ALL_INSTANCES	All instances of SBI_VVC (constant = -2)
5.	VVC_BROADCAST, ALL_INSTANCES	All instances of all VVCs

Figure 6: Target options for a channel based VVC

Commands can target a single VVC, all instances and channels of a VVC, or all VVCs in the test environment, as listed in Figure 6.
A single VVC is targeted using its instance number, and with its channel name if applicable. Alternatively, all instances or channels of a VVC can be targeted using the ALL_INSTANCES or ALL_CHANNELS keywords, respectively.
The VVC_BROADCAST keyword is used when targeting all of the VVCs in the test environment, e.g. when enabling or disabling messaging, flushing command queues or synchronizing VVC command executions.

2.3.3 Queuing

The only functional difference between calling a BFM (from inside the central sequencer) vs a CDM - seen from a black box point of view, is that the CDM will have to wait in a queue locally inside the VVC until all previously entered commands in that queue have been executed. If no command is pending (in the queue) and no command is currently being executed towards the DUT via this VVC, then the CDM and BFM behave exactly the same.

The command distribution from the sequencer to the VVCs explained above means the sequencer may distribute commands to multiple VVCs at the same time. This because the actual distribution of commands is not consuming any time, but happens instantaneously. This allows the sequencer to initiate accesses on several DUT interfaces simultaneously.

For BFMs another BFM-call would not have been possible at all from the sequencer, and would thus have blocked the sequencer from doing anything else. Process-based BFMs might have allowed queuing of commands, but often with a terrible overview of what is actually happening in the system.

The queuing mechanism inside the VVC allows the sequencer to distribute (again in zero time) a sequence of commands to any given VVC for back to back queued execution.

Every single CDM is given a unique command index, counting from 1 upwards for every CDM called from the central test sequencer. The actual index for a given command is available by executing *'get_last_received_cmd_index (vvc_target, vvc_instance, [vvc_channel,], [msg])'* immediately after distribution of that command. This index may be used for various purposes by the sequencer. One example could be to fetch the result of a CDM, e.g. for a read-command, to check if a command has been executed, and to wait for a given command to complete. The latter is handled by the CDM 'await_completion()'. This CDM will stall the sequencer until a previous indexed CDM (or

all previous CDMs) to a given VVC has been executed on that VVC. This mechanism is excellent for synchronization of events inside the testbench.

2.3.4 Test sequencer example

We can illustrate the test sequencer operation by considering a UART testbench as shown in Figure **4** or Figure **5**. Note that the VVCs are emulating the environment and thus the VVC Tx channel is connected to the DUT Rx.

Now let us interpret the test sequencer example shown in Figure 7.

We can see that all the procedure calls are CDMs, i.e. distribution of commands to the VVCs. This can be seen directly from the command syntax - as all the procedure calls start by specifying the target in the first parameters. In the figure the targets have been marked as red to clearly differentiate between target parameters and the other following parameters.

Figure **8** shows the timing diagram for the VVCs execution activity and the interface towards the DUT. Please note the spacer symbols in the figure, and that the access time relations are not as indicated by the widths shown in the figure. (E.g. the SBI access is in reality much shorter compared to the UART access.)

Simple test sequencer example for the UART TB:

```
1   sbi_write(         SBI_VVCT,1,      C_ADDR_TX, x"2A", "Uart TX");
2   uart_expect(       UART_VVCT,1,RX,  x"2A", "From DUT TX");
3   uart_transmit(     UART_VVCT,1,TX,  x"C1", "Into DUT RX");

4   insert_delay(      UART_VVCT,1,TX,  2*C_BIT_PERIOD);
5   uart_transmit(     UART_VVCT,1,TX,  x"C2", "Into DUT RX");

6   await_completion( UART_VVCT,1,RX   );
7   await_completion( UART_VVCT,1,TX   );

8   sbi_check(         SBI_VVCT,1,      C_ADDR_RX, x"C1", "Uart RX");
9   sbi_check(         SBI_VVCT,1,      C_ADDR_RX, x"C2", "Uart RX");
10  await_completion( SBI_VVCT,1       );
```

Figure 7: UART TB test sequencer

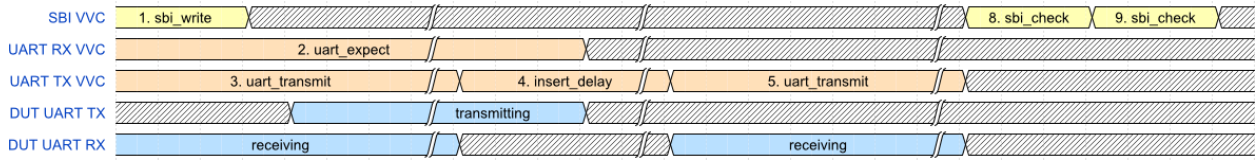

Figure 8: Timing diagram for Simple test sequencer example above

On lines 1-3 in the example the sequencer is distributing a single command to each of three different destinations, SBI_VVC 1, UART_VVC 1 RX and UART_VVC 1 TX. This distribution is non time consuming. All the "leaf-VVCs" have now received one command each, and will start execution immediately.

Line 4 - insert_delay() – is put into the execution queue for UART_VVC,1,TX after the transmit command given in line 3. Then another transmit command (line 5) is distributed to the same queue. Hence after line 5 the queue inside UART_VVC,1,TX has 3 commands pending (lines 3,4,5).

On line 6 the sequencer stops running non time consuming commands as it initiates await_completion(). This CDM is not allowed to finish until UART_VVC,1,RX has executed all pending commands towards the DUT, i.e. until uart_expect() has completed. This of course is a time consuming command, executed as a BFM from the VVC towards the DUT. And once time is starting to run, all queued commands will execute – in parallel if on different interfaces, or in order if on the same interface.

In the timing diagram this can be seen as immediate activity on all VVC interfaces. SBI_VVC and UART_TX_VVC start transmission immediately (initiated by lines 1 and 3), while UART_RX_VVC starts waiting for data immediately (initiated by line 2), and receiving data soon after the SBI_WRITE is completed.

As soon as UART_VVC,1,RX has completed its byte reception, it is finished – as there are no more commands in its queue. This corresponds to the end of the uart_expect transaction in the timing diagram. The await_completion() command is then allowed to finish and the sequencer may continue to line 7. UART_VVC,1,TX will wait for 2*C_BIT_PERIOD from completion of the first transfer to the start of the next – due to the insert_delay() command. When the second transmit is completed the sequencer is allowed to continue to line 8.

At this stage we know that there is no more pending activity in the UART VVC, and that one byte has been received and two bytes transmitted. We also know that the sbi_write() (line 1) has been executed – as otherwise the uart_expect() would have failed.

Finally two sbi_check() commands are distributed to SBI_VVC,1 to check that the two bytes from lines 3 and 5 have been successfully received. They should now be available in the UART receive buffer of the DUT – ready to read via the CPU interface.

Again the distribution of commands is non time consuming until the await_completion() in line 10, which doesn't finish until both sbi_check() commands have been executed.

The sequencer itself does not perform any checks in this example. It just distributes commands to the VVCs and allows them to handle the command executions autonomously. Thus the VVCs will do the requested checking and potentially write a positive acknowledge to the log and simulation transcript. If the check fails the VVC will scream out loud and stop the simulation if set up to do so.

2.4 Test sequencer considerations

The above test sequencer example was of course just a very small piece of code to illustrate how to read and understand the sequence of events.

The example code would be part of a test sequencer process with local declarations and potentially an initial setup section. An example of a complete testbench and test sequencer can be found for the UART in 'bitvis_uart/tb/uart_vvc_tb.vhd'.

An advanced testbench for a complex DUT would typically have more advanced procedures handling verification at a higher level, but the example shown in this document and in the provided example is intended as a simple example on using VVC FRAMEWORK and its provided functionality.

It is generally recommended to stick to one single central sequencer – as a single "brain" in a system is almost always easier to follow and understand. It is however possible to have multiple central sequencers if you like. They can always use await_completion() to synchronize and align, but they could also use the built in direct synchronization methods from Utility Library (block|unblock|await_unblock_flag and await_barrier)

2.5 Sequencer direct access to VVC configuration and status

The configuration and access records given in the quick references are directly available from the sequencer – as shared variables.

Hence the sequencer may configure a VVC directly as

```
shared_<vvc-name>_config(instance-num).<field-name> := <whatever>;
```

```
e.g. shared_sbi_vvc_config(1).clock_period := 10 ns;
```

And status may be read directly as

```
<variable/signal>  := shared_<vvc-name>_status(instance-num).<field-name>;
```

```
e.g.  my_integer := shared_sbi_vvc_status(2).current_cmd_idx;
```

3 Using the VVC Framework

3.1 Prerequisites

It is strongly recommended before you commence that:

1. You understand the overall concepts and functionality of the UVVM Utility Library. See uvvm_util/doc
2. You have understood the previous section (2) in this document 'Understanding the VVC Framework'.

3.2 Making your own testbench architecture

As explained in section 2.2 the architecture may be implemented in a very structured and simple manner – with a good overview.

First make your normal simple testbench and simple test harness as you like – as a starting point. Then all you have to do to structure it properly using VVCs in a good testbench architecture, is to connect each VVC to the corresponding interface on the DUT – as any other inter entity (or component) connection. Then you assign values to the generics of your VVC instantiations wherever the default is not wanted.

Note that you need to instantiate 'uvvm_vvc_framework.ti_uvvm_engine' in your testbench, and you should include wait_for_uvvm_init() as your first statement in your test case sequencer. You do of course need to include the necessary libraries.

(See 'bitvis_uart/tb/uart_vvc_th|tb.vhd' as examples)

3.3 Making your own VVC FRAMEWORK test sequencer

You must of course know which VVCs are connected to your DUT. This you can find out by looking at the testbench architecture, or you can start running your testbench (even without a sequencer) and it will report all connected VVCs, their instance numbers and channel (if applicable), provided constructor messages have not been disabled.

Then all you have to do is to call a sequence of CDMs with relevant parameters – as shown in Figure 7 or in the UART example in 'bitvis_uart/tb/uart_vvc_tb.vhd'.

You can find all available CDMs in the quick references for VVC FRAMEWORK methods (common methods for all VVCs) and for each individual VVC. If you are using non-official VVCs (your own or third party) a quick reference may not be available. If so you can find the methods under <vvc-directory>/src/vvc_methods_pkg.vhd.

If something doesn't work as expected – turn on more verbosity (see chapter 4)

3.4 Making your own VVC and VVC methods

Remember that it is always assumed that you have all the required BFM procedures available prior to making a VVC. These procedures are critical for any type of testbench, and should thus always be implemented at an early stage in the verification process.

To make your own VVC then first run the Python script uvvm_vvc_framework\script\vvc_generator\vvc_generator.py.

This will generate a new VVC based on a non-channel or channel based VVC depending on your selection. Then go through the generated files and make the necessary modification. Please see uvvm_vvc_framework\doc\VVC_Implementation_Guide.pdf for information on the various files.

If something doesn't work as expected – turn on more verbosity (see chapter 4).

3.5 Library and package hierarchy for VVCs

This chapter is only meant to be read if you really need to understand the details of the system. It is not at all needed for anyone just writing testcases (test sequencers), and for VVC designers it is only of interest if you want to understand the exact relation between the various VHDL packages. This section requires good VHDL knowledge.

Any VVC is based on a VVC entity with an interpreter, a queue and an executor as the main command handling blocks. To simplify understanding and re-use, most implementation details are located in packages. These packages may basically be divided into three categories.

1. 'VVC dedicated packages' (functions, procedures, types, constants, global signals and shared variables):
   Functionality that is dedicated for a given VVC, where the implementation is targeted at the needs of this specific VVC. E.g. the uart_receive CDM and the shared_vvc_cmd containing all UART_VVC specific record fields.
   - Such packages are located under the relevant VVC and are compiled to the library of that VVC.
   - Marked as light yellow in Figure 9.
2. 'VVC Framework Library':
   Functionality that is common for all (or most) VVCs, and is independent of VVC dedicated definitions/declarations.
   - Such packages are located under the UVVM_VVC_Framework directory and are compiled to the uvvm_vvc_framework library.
   - Shown partly Figure 9 in blue. These two packages are referenced by lots of other packages in the VVC libraries. There are more packages in this library, but these are only referenced by these two packages, and not by the VVC related packages.
3. 'VVC FRAMEWORK target dependent packages'
   Functionality that is common for all (or most) VVCs, but is dependent on VVC dedicated definitions/declarations.
   - Such packages are located under the UVVM_VVC_Framework directory as their contents are common, but they are compiled into VVC libraries as they depend on other compiled packages in their respective VVC library.
   - Shown in Figure 9 as orange – to indicate that the packages are located under UVVM_VVC_Framework, but compiled into a dedicated VVC

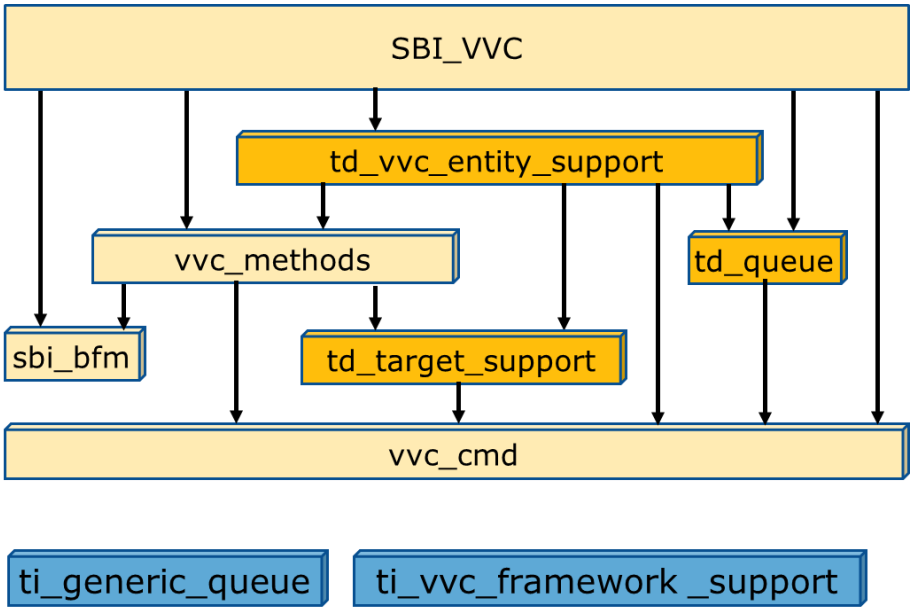

Figure 9: VVC Package organisation

The two first package categories are quite normal in any system – with local and common support for various functionality. The third package category is a bit different. The actual files and thus package contents are the same across all VVCs - for the simple reason that they all need the same functionality support - like for instance procedure 'fetch_command_and_prepare_executor()' inside 'td_vvc_entity_support_pkg'. However, as the actual commands are specific to each individual VVC, and this procedure is fetching these commands, the command type must be known for the procedure and thus also for the package. Hence 'td_vvc_entity_support_pkg' must reference 'vvc_cmd_pkg' in which the command type is defined for this specific VVC. For 'td_vvc_entity_support_pkg' to be single source for all VVCs, this package must reference 'vvc_cmd_pkg' in its own local library (work). Thus they must both be compiled into the same VVC library.

You will find that the VVC Framework packages that are target independent – i.e. as normal support packages, are located under the src directory as you would expect. These packages are compiled into the UVVM_VVC_framework library as they are used as common support files for the complete system. These packages have been prefixed with 'ti' to indicate that they are target independent.

The VVC Framework packages that are target dependent – i.e. common support packages that depend on VVC-dedicated declarations in a VVC library, are located under directory 'src_target_dependent' to clearly show that these packages are different. These packages are compiled into all VVC libraries and reference for instance the 'vvc_cmd_pkg' available in the that library. These packages have been prefixed with 'td' to indicate that they are target dependent.

Note that most packages and components reference the UVVM Util Library and UVVM VVC Framework library for common functionality. The dependency on these libraries are not shown in the figures – to simplify the overview.

3.6 Library and package hierarchy for the central test sequencer

The central test sequencer(s) must have access to all available methods for every VVC in the testbench. The package 'vvc_methods' provides all the VVC dedicated methods for that VVC.

Figure 9 shows all the packages needed for the VVC to compile, whereas Figure 10 shows all packages compiled into the VVC library.
'td_vvc_framework_common_methods' is a package located under the uvvm_vvc_framework directory – as the code is common for all VVCs, but it is compiled into each VVC because it depends on declarations in each specific VVC library.

For every VVC the sequencer must include both 'vvc_methods' and 'td_vvc_framework_common_methods' to get access to both VVC dedicated and general commands for each VVC. Figure 11 shows that for a test harness with three VVCs A, B and C, the sequencer must include 3*2 packages.

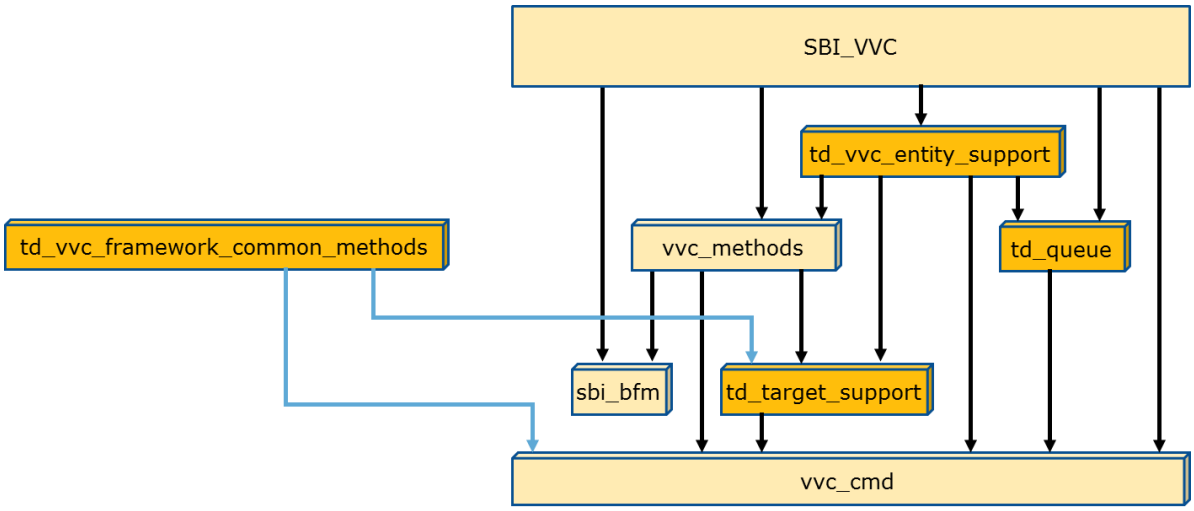

Figure 10: Packages in VVC library

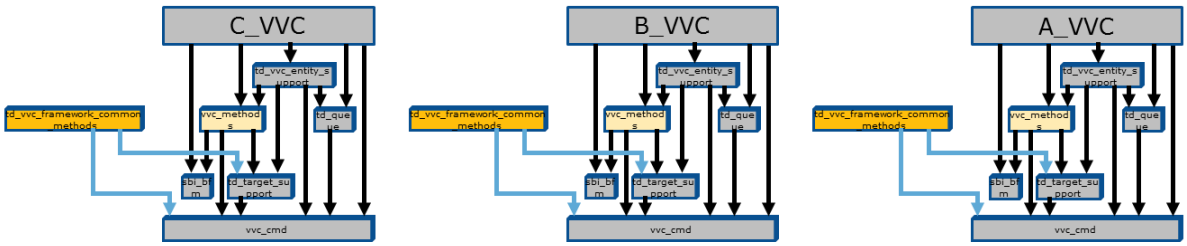

Figure 11: Packages referenced by central sequencer

4 Debugging

The example VVCs, BFMs and Testbench show how one should always write log messages to allow good simulation progress reports, and write checks to provide good alert handling and mismatch reports.

These mismatch reports and log messages are vital when trying to debug your design or testbench.

4.1 Increasing the verbosity

There are several ways of increasing the verbosity of your testbench – provided you have followed the examples of the attached VVC FRAMEWORK example testbench under 'bitvis_uart/tb/'.
(For targets see section 2.3.2, For IDs see 'UVVM_Util/src/adaptations_pkg.vhd')

1. Controlling verbosity via the test sequencer
    a. Brute force:
    enable_log_msg(<target>, ALL_MESSAGES), or
    disable_log_msg(<target>, ALL_MESSAGES)
    b. Selected:
    enable_log_msg(<target>, <ID>), or
    disable_log_msg(<target>, <ID>)
    c. Any combination of the above
2. Controlling verbosity via default setup in 'UVVM_Util/src/adaptations_pkg.vhd'

    constant C_VVC_MSG_ID_PANEL_DEFAULT - for all VVCs and
    constant C_MSG_ID_PANEL_DEFAULT - for all sequencer logs

In general it is a good idea to have maximum verbosity when starting to develop a testbench or a VVC.

Hint: It might be a good idea to always run with a high verbosity, and then just filter on the log after simulation. The IDs and Scopes yield excellent filtering opportunities.

4.2 Recommended verbosity

4.2.1 For regression tests

Enable log headers only – as they should reflect your specification

4.2.2 For simple overview on sequence of events – but not debugging

Keep only log headers and a single occurrence of any command

Alt.1: ID_LOG_HDR(*1) + ID_SEQUENCER + ID_BFM/IMMEDIATE_CMD in every VVC

Alt.2: ID_LOG_HDR(*1) + ID_SEQUENCER + ID_UVVM_SEND_CMD

Alt.3: Both above.

(*1) : ID_LOG_HDR, ID_LOG_HDR_XL, ID_LOG_HDR_LARGE depending on your usage.

4.2.3 For detailed debugging

The simplest alternative is to turn on all verbosity for the problem at hand:

E.g. full global verbosity (not specifying any VVC) and full verbosity for the relevant VVCs. Full verbosity is set using a special ID of 'ALL_MESSAGES'.

If this is too much, either try to disable irrelevant IDs or do it all the other way around by starting with alt. 3 in the previous chapter and enable more IDs as required.

5 License

See license info in the download.

5.1 License opportunities

As UVVM is using the rather relaxed Apache license there are multiple options available for the VHDL community or vendors.

You may develop your own VVCs or add-ons and either:

- Keep it internally with no need to publish
- Publish as open source – free or commercial
- Give away or Sell to anyone you like – as IP or as a part of a delivery
- etc…

Given of course that you comply with the Apache license.