UVVM VVC Framework Essential Mechanisms – Quick Reference

This document explains some of the essential mechanisms necessary for running VVC Framework, in addition to helpful and important VVC status and configuration records which are accessible directly from the testbench. More details on the VVC Framework and the command mechanism can be found in the VVC Framework Manual.

VHDL 2008 only

1 Libraries

In order to use a VVC the following libraries need to be included:

```
library uvvm_util;
context uvvm_util.uvvm_util_context;

library uvvm_vvc_framework;
use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;

library bitvis_vip_<name>;
context bitvis_vip_<name>.vvc_context;
```

2 UVVM Initialization

The following mechanisms are required for running UVVM VVC Framework

Mechanism	Description
ti_uvvm_engine	**ti_uvvm_engine** This entity contains a process that will initialize the UVVM environment, and has to be instantiated in the testbench harness, or alternatively in the top-level testbench. Example: `i_ti_uvvm_engine : entity uvvm_vvc_framework.ti_uvvm_engine;`
await_uvvm_initialization()	**await_uvvm_initialization(VOID)** This procedure is a blocking procedure that has to be called from the testbench sequencer, prior to any VVC calls, to ensure that the UVVM engine has been initialized and is ready. This procedure will check the shared_uvvm_state on each delta cycle until the UVVM engine has been initialized. Note that this method is depending on the ti_uvvm_engine mechanism. Note that this method uses the `t_void` parameter, defined in the UVVM Utility Library types package. Example: `await_uvvm_initialization(VOID);`

3 UVVM and VVC user accessible shared variables and global signals

UVVM and VVC shared variables and global signals are defined in global_signals_and_shared_variables_pkg and the various VVC packages.

Shared variables

Shared variable	Description

shared_uvvm_status	Shared variable providing access to VVC related information via the `info_on_finishing_await_any_completion` record element, i.e. `shared_uvvm_status.info_on_finishing_await_any_completion`
	
	This record element gives access to the name, command index and the time of completion of the VVC that first fulfilled the `await_any_completion()`. The available record fields are:
	
	`vvc_name : string -- default "no await_any_completion() yet"`
	`vvc_cmd_idx : natural -- default 0`
	`vvc_time_of_completion : time -- default 0 ns`
	
	For more information regarding other fields available in the `shared_uvvm_status` see the UVVM Util QuickRef, section 1.4

shared_<vvc_name>_vvc_config	Shared variable providing access to configuration parameters for each VVC instance and channel if applicable. E.g.
	
	`shared_sbi_vvc_config(1).inter_bfm_delay.delay_type := TIME_START2START;`
	`shared_uart_vvc_config(RX,1).bfm_config.bit_time := C_BIT_TIME;`

shared_<vvc_name>_vvc_status	Shared variable providing access to status parameters for each VVC instance and channel if applicable. E.g.
	
	`v_num_pending_cmds := shared_sbi_vvc_status(1).pending_cmd_cnt;`
	`v_current_cmd_idx := shared_uart_vvc_status(TX,2).current_cmd_idx;`
	`v_previous_cmd_idx := shared_uart_vvc_status(TX,2).previous_cmd_idx;`

global_<vvc_name>_vvc_transaction_trigger	Global trigger signal for when a VVC has updated its shared variable with VVC transaction info. See section 6 for more details.

shared_<vvc_name>_vvc_transaction_info	Shared variable providing access to Transaction Info VVC instances. See section 6 for more details.
	Available information is dependent on VVC type and typical information is:
	
	`operation : t_operation; -- default NO_OPERATION`
	`data : std_logic_vector(C_VVC_CMD_DATA_MAX_LENGTH-1 downto 0); -- default 0x0`
	`vvc_meta : t_vvc_meta; -- default C_VVC_META_DEFAFULT`
	`transaction_status : t_transaction_status; -- default C_TRANSACTION_STATUS_DEFAULT (INACTIVE)`
	
	Note that this shared variable is replacing the shared_<vvc_name>_transaction_info, which will soon be deprecated.

4 VVC Status, Configuration and Transaction information

The VVC status, configuration and transaction information records are defined in each individual VVC methods package.

Each VVC instance and channel can be configured and useful information can be accessed from the testbench via dedicated shared variables.

From the VVC configuration shared variable, one is given the ability to tailor each VVC to one's needs, in addition to access the BFM configuration record via the `bfm_config` identifier. In addition to BFM configuration possibility, the configuration settings consist of command and result queue settings, BFM access separation delay and a VVC dedicated message ID panel.
Note that some BFMs require user configuration, e.g. the `bit_time` setting in serial interface BFMs.

The VVC status shared variable provide access to the command status parameters for each of the VVCs, such as the current and previous command index, and the number of pending commands in the VVCs command queue. This provide a helpful tool, e.g. when synchronizing VVCs in the test sequencer using the `await_completion()` or `await_any_completion()` methods.

When using a wave viewer during simulation, the transaction shared variable provides helpful information regarding current VVC operation and transaction information such as `address` and `data`. Note that the accessible fields depend on the VVC and its implementation. An example of two SBI VVCs performing FIFO write operations, followed by check operations, is shown in Figure 1.

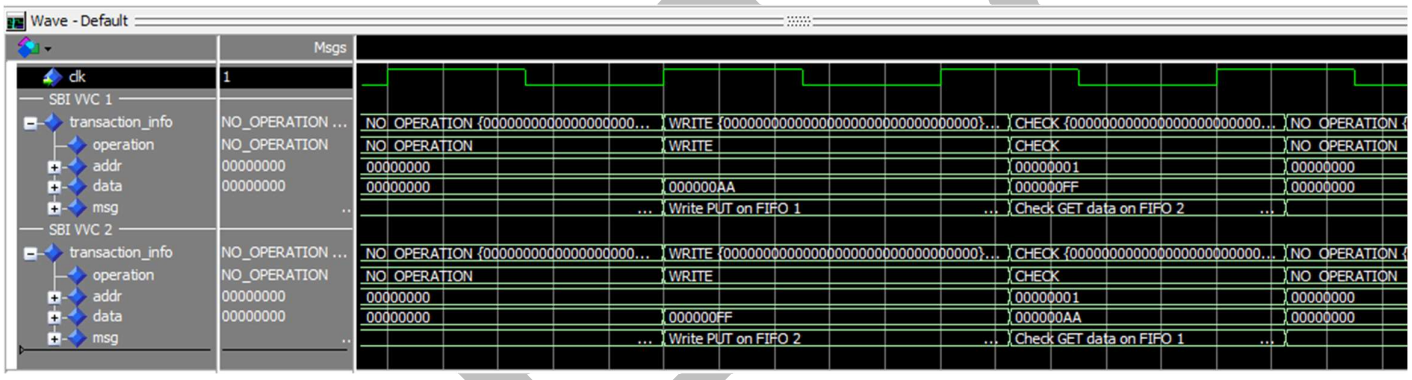


Figure 1 VVC Transaction info example

5 Activity Watchdog

UVVM VVC Framework has an activity watchdog mechanism which all Bitvis VVCs support. All VVCs can be automatically registered in a centralized VVC activity register at start-up and will, during simulation, update the VVC activity register with their current activity status, i.e. ACTIVE or INACTIVE, which again is monitored by the activity watchdog. A timeout counter in the activity watchdog will start after the last update has occurred in the VVC activity register, and the timeout counter is reset on any VVC activity. An alert will be raised if none of the VVCs have an activity prior to the timeout counter reaching the specified timeout value. Note that the activity watchdog will continue to monitor VVC activity, even after a timeout, until alert stop limit is reached.

The activity watchdog has to be included as a concurrent procedure parallel to the testbench sequencer or in the test harness in order to be activated.
Note that the activity watchdog will raise a TB_WARNING if the number of expected VVCs does not match the number of registered VVCs in the VVC activity register, and that leaf VVCs (e.g. channels such as UART RX and TX) are counted individually. This checking can be disabled by setting the number of expected VVCs to 0.
Also note that the total number of VVCs registered in the VVC activity register cannot exceed the `C_MAX_TB_VVC_NUM` count, default set to 20 in the `adaptations_pkg.vhd`, and this will result in a TB_ERROR raised by the VVC activity register.

Note that some VVCs should not be monitored by the activity watchdog. This currently applies to the clock generator VVC, as this VVC may continue to be active even after all other testbench activity has stopped. This VVC will have to be included in the number of expected VVCs registered in the VVC activity register but will not have any effect on the activity watchdog timeout counter.

Field name	Type name	Default	Description
num_exp_vvc	natural		Expected number of VVCs which is expected to be registered in the VVC activity register (including any clock generator VVC). Note 1 : each channel is counted as an independent VVC expected to be registered in the VVC activity register. Note 2 : setting num_exp_vvc = 0 will disable checking of number of expected VVCs vs actual number of VVCs registered in the VVC activity register
timeout	time		Timeout value after last VVC activity.
alert_level	t_alert_level	TB_ERROR	The timeout will have this alert level.
msg	string	"Activity_Watchdog"	Message included with activity watchdog log messages, e.g. name of activity watchdog

Example:

```
p_activity_watchdog:
    activity_watchdog(num_exp_vvc   => 3,
                      timeout       => C_ACTIVITY_WATCHDOG_TIMEOUT);
```

6 Distribution of Transaction Info – From VVCs and/or Monitors

UVVM now supports sharing transaction information in a controlled manner within the complete testbench environment. This allows VVCs and Monitors to provide transaction info to any other part of your testbench – using a predefined structured mechanism. This makes it even easier to make good VHDL testbenches.

Transaction information may be used in many different ways, but the main purpose is to share information inside the testbench of activity or accesses on a given DUT interface. Such information could typically be provided from a dedicated interface monitor, but making such a dedicated monitor is sometimes quite time consuming and often not really needed. For that reason, UVVM provides a mechanism for getting the transaction information directly from the VVC.

6.1 Purpose

By enabling the distribution of transaction info, models or any other parts of the testbench can see exactly what accesses have been made on the various interfaces of the DUT, so that the expected DUT behaviour and outputs may be checked. Let us illustrate this with a really simple testbench scenario to verify a UART peripheral with an AXI-lite register/CPU interface on one side and the UART RX and TX ports on the other side. The test sequencer may command the AXI-lite BFM or VVC to write a data byte into the UART TX register, and then it must be checked that the data byte is transmitted out on the DUT TX output some time later.

a) A simple testbench approach could be to have the test sequencer also telling the receiving UART BFM or VVC exactly what to expect. This is a straightforward approach, but requires more action and data control inside the test sequencer. This could of course all be handled in a super-procedure, but for any undetermined behaviour inside the BFM or VVC, like random data generation or error injection, that would not work. See Figure 2.
b) A more advanced approach is to have a model overlooking the DUT accesses, generate the expected data and tell the receiving BFM or VVC to check for that data. See Figure 3.
c) An even more advanced approach would be to use a Scoreboard to check received data (from DUT via VVC) against expected data from a model. See Figure 4.

However, for the two latter approaches the model needs information about exactly what happened (the transaction) on the various DUT interfaces, so that it can generate the correct expected data. For the model it doesn't matter if the transaction info comes from a monitor or from a VVC, as long as the information is correct.

The model could of course look at the interfaces and analyse the actual transactions, but distributing this task to the VVC or monitor makes the testbench far more structured and significantly improves overview, maintenance, extensibility and reuse – at least for anything above medium simple verification challenges.

Another purpose of providing transaction information is for progress viewing and debugging – typically via the wave view or simulation transcripts.

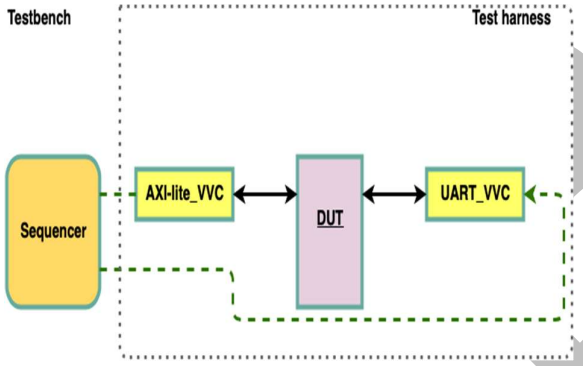


Figure 2 Distribution of Transaction Info Approach A

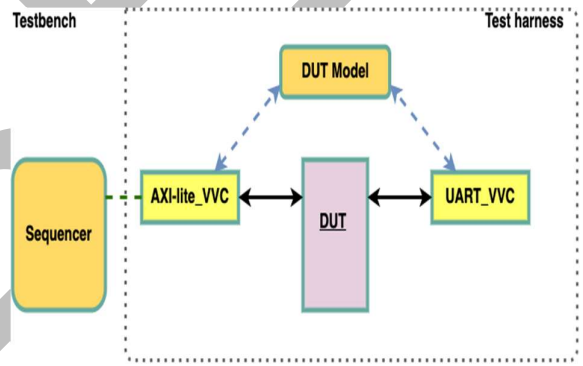


Figure 3 Distribution of Transaction Approach B

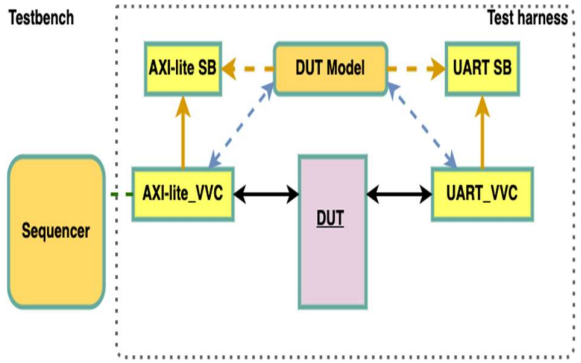


Figure 4 Distribution of Transaction Approach C

6.2 Transaction definitions

By transactions we normally talk about a complete end to end transfer of data across an interface. This could be anything from a simple write, read or transfer of a single word - to a complete packet in a packet-oriented protocol like Ethernet. The word transaction is however also used for both sub-sets and super-sets of this – depending on the protocol and even on how we want to control our system or testbench. In order to communicate properly and to assure that transactions are properly understood, the following terms are defined:

- **Base transaction (BT)** is the lowest level of a complete transaction as allowed from the central sequencer.
  E.g. AXI-lite, UART, Ethernet and AXI-Stream transactions.
- **Sub-transaction (ST)** is the lowest level of an incomplete transaction as allowed from a BFM. The sub-transaction as such is complete seen from a handshake point of view, but the transfer of data is not complete. A split transaction protocol will typically have sub-transactions.
  E.g. Avalon Read Request and Avalon Read Response.
- **Leaf transaction (LT)** is not a transaction type in itself, but is the lowest level of complete **or** incomplete transaction defined for a given protocol. I.e. a sub-transaction when this is defined for a given protocol, otherwise a base transaction. This definition is needed in order simplify various explanations.
  (e.g. for Avalon: LT = the sub transactions, and for UART, SBI or Ethernet: LT = the base transactions (as no sub-transaction exist for these protocols)
- **Compound transaction (CT)** is a set of transactions or other methods or statements that as a total is doing a more complex operation. A compound transaction involves calling multiple base transactions. E.g. SBI_Poll_Until() or a UART transmit of N consecutive bytes. No compound transaction needs to be defined.

6.3 Transaction information

Information about the above transactions is typically provided to a model in the test harness. Depending on whether the transaction info is provided from the VVC or Monitor, different types of information will be available. Common for both is that they always provide info about the operation (read, write, transmit, etc) and often also any other protocol specific info. For a UART this could be data and parity, for an SBI it could be address and data, and for Ethernet: the packet fields.
This minimum is normally what the Monitor can provide from just analysing the interface, and this is also normally enough for a model to generate expected DUT outputs. The VVC on the other hand, can provide more info, which could be useful for instance for progress viewing and debugging.
The transaction information is organised as a transaction record with some predefined fields as shown below. The first table shows the general transaction record, whereas the second table shows a concrete example for the SBI.
Note that for a given interface/protocol, the VVC and the Monitor will use the same interface dedicated transaction record type – with some fields potentially unused.

Table 1 – General Transaction record t_transaction. The greyed-out fields indicate optional or protocol dedicated fields

Field name	Type name	Description
Operation	t_operation	Protocol operation on the given DUT interface. E.g. NO_OPERATION, WRITE, READ, TRANSMIT, POLL_UNTIL, … NO_OPERATION is default and thus used when there is no access. All operations will be separated with a NO_OPERATION for at least 1 delta cycle, e.g. NO_OPERATION – WRITE – NO_OPERATION – READ – NO_OPERATION.
<Optional protocol dedicated field(s)>	<Protocol dedicated>	One or more fields required to complete the transaction info E.g. for UART: Single field 'data'; for SBI: field 1: 'addr', field 2: 'data'; for Ethernet: Most ethernet fields as separate fields here – or a better solution include as a complete sub-record
Transaction_status	t_transaction_status	Transaction status. Handled slightly different from a VVC and a Monitor. VVC: Will show 'IN_PROGRESS' during the transaction and INACTIVE in between (for at least one delta cycle) Monitor: Will show FAILED or SUCCEEDED immediately as soon as this is 100% certain – and keep this info for the display period defined in the Monitor configuration record, or until the start of the next transaction, whichever occurs first. Other than that it will show IN_PROGRESS when active or INACTIVE when not.
VVC_Meta	t_vvc_meta	Additional transaction information – only known by the VVC. So far 'msg' and 'cmd_idx' (the free running command index) A monitor has no knowledge of this as and will set them to msg = "", cmd_idx = -1
Error_info	t_error_info	Any type of error injection relevant for the given protocol. Typically parity or stop-bit error in an UART or a CRC error in an Ethernet. If no error injection or detection has been implemented, this sub-record may be left out.
NOTES: - For transaction info from a VVC the record reflects the command status, i.e. the status assumed by the VVC when initiating the command, whereas the Monitor will set up the record only after knowing whether the transaction has failed or succeeded. The VVC does not know the BFM status, and this is fine because the BFM will issue an alert for unexpected behavior.		

Table 2 – SBI specific Transaction record t_transaction. The greyed-out fields indicate protocol dedicated fields

Field name	Type name	Description
Operation	t_operation	Either of WRITE, READ or CHECK, but could also be POLL_UNTIL or a more complex compound transaction
Address	unsigned	
Data	std_logic_vector	
Transaction_status	t_transaction_status	
VVC_Meta	t_vvc_meta	
Note: No error_info field as no error injection or detection has been implemented in neither VVC nor Monitor – at this stage.		

Other interfaces will of course have different protocol dedicated fields, or even a complete protocol dedicated sub-record (e.g. for Ethernet packet fields)

6.4 Transaction info transfer

In order to reduce the number of signals from a VVC or Monitor, all possible simultaneous transactions (and their transaction records) are collected into a single transaction group record. For an SBI interface this will consist of a BT record and potentially a CT record, whereas for an Avalon it will in addition also consist of two ST records because for instance a read request may be active at the same time as a read response. (And the sub-transactions are part of a base transaction and may also be part of a CT).

Table 3 below shows the maximum transaction group record for an SBI, whereas Table 4 below shows the maximum transaction group record for an Avalon. The greyed-out CT is optional for both, and thus depends on whether CTs have been defined in the VVC. Multiple parallel STs may be written to the transaction group record simultaneously – as these are handled by different “threads” (concurrent statements like a process).

A Monitor cannot know about CTs, and thus a monitor will never fill inn that sub-record. A Monitor for a split transaction protocol (i.e. with multiple STs) may or may not provide BT info. If it does, this should normally be implemented in a higher level “wrapper”

Note again:
- A VVC will update its Transaction Info leaf transaction details at the start of the transaction when the BFM is called. and turned off when BFM is finished.
- A monitor will set its Transaction Info record after the transaction is finished (or transaction status is known) and keep it on for a pre-defined time – or until the next transaction is finished if earlier.

It is recommended that the model (or any other user of Transaction Info) trigger on the VVC/monitor trigger signal and check when transaction_status is changing to ‘INACTIVE’ and then sample <signal>‘last_value

Table 3 – Maximum transaction group record t_transaction_group – for an SBI interface

Field name	Type name	Description
bt	t_transaction	Base transaction
ct	t_transaction	Compound transaction

Table 4 - Maximum transaction group record t_transaction_group – for an Avalon MM interface

Field name	Type name	Description
st_request	t_transaction	Sub-transaction
st_response	t_transaction	Sub-transaction
bt	t_transaction	Base transaction
ct	t_transaction	Compound transaction

6.5 Transaction Info Record signals

The Transaction Info Record is provided out of the VVC and Monitor using sets of a global signal and a shared variable. These and all Transaction Info related VHDL types are defined in transaction_pkg, located in the VIP src folder.

- Monitor trigger signal : global_<protocol-name>_monitor_transaction_trigger, e.g. global_uart_monitor_transaction_trigger
- Monitor shared variable : shared_<protocol-name>_monitor_transaction_info, e.g. shared_uart_monitor_transaction_info
- VVC trigger signal: global_<protocol-name>_vvc_transaction_trigger, e.g. global_uart_vvc_transaction_trigger.
- VVC shared variable : shared_<protocol-name>_vvc_transaction_info, e.g, shared_uart_vvc_transaction_info. The VVC is also responsible for filling out the vvc_meta record field.

*Table 5 – Transaction Info record **'t_<if>_transaction_group'** for an UART interface -- accessible via **shared_uart_vvc_transaction_info(vvc_idx)** t_uart_transaction_group_array*

Record element	Type	Default	Description
Bt	t_transaction	C_TRANSACTION_SET_DEFAULT	Transaction Info record entry for base transaction.
→ operation	t_operation	NO_OPERATION	Equal to VVC transaction operation, e.g. TRANSMIT, RECEIVE and EXPECT (UART)
→ address [1]	unsigned	0x0	DUT access read or write address.
→ data	std_logic_vector	0x0	DUT read or write data.
→ vvc_meta	t_vvc_meta	C_VVC_META_DEFAULT	Record of meta data belonging to VVC command request resulting in this base transaction.
→ msg	string	" "	Message transmitted with VVC command resulting in this base transaction.
→ cmd_idx	integer	-1	VVC command index resulting in this base transaction.
→ transaction_status	t_transaction_status	C_TRANSACTION_STATUS_DEFAULT	The current status of transaction. Available statuses are INACTIVE, IN_PROGRESS, FAILED and SUCCEEDED. [2]
→ error_info [3]	t_error_info	C_ERRO_INFO_DEFAULT	Record entry of errors that will be injected to the DUT access transaction.
→ parity_bit_error[4]	boolean	False	The DUT transaction will have a parity bit error if entry is set to *true*.
→ stop_bir_error [4]	boolean	False	The DUT transaction will have a stop bit error if entry is set to *true*.
ct [5]	t_transaction	C_TRANSACTION_SET_DEFAULT	Transaction Info record entry for compound transaction. Note that sub-record entries would typically have the same entries as for a base transaction, and that this entry does not have to be suited for all interface Transaction Info records.

Note 1: record field ***address*** is not applicable for all interface types, e.g. UART, and is only shown here for informational purposes.
Note 2: transaction status ***FAILED*** and ***SUCCEEDED*** are not applicable for VVC Transaction Info records, but will be used for Monitor Transaction Info records.
Note 3: record field ***error_info*** and its sub-record fields can be omitted if no error injection is implemented in the BFM.
Note 4: ***error_info*** sub-record fields ***parity_bit_error*** and ***stop_bit_error*** are examples of UART error injection.
Note 5: record entry ***ct*** will consist of similar record fields as ***bt***, and might not always be necessary. This applies to record entry ***st*** as well (not shown here).

6.6 Transaction Info Signal and Shared Variable

A VVC or Monitor will trigger the global trigger signals listed in section 6.5 when information of a new transaction info is made available. For a VVC the transaction info is made available prior to the corresponding bus access, i.e. before calling the BFM method, and the global trigger signal will be pulsed for a delta cycle. The VVC will set the transaction info back to default values immediately after the bus access has finished, but then without pulsing the global trigger signal.
For a Monitor the transaction info will be made available immediately after a bus access is completed and then the global trigger signal will be pulsed for a delta cycle. The transaction info is valid when the global trigger signal is pulsed and is set back to default values after a period of transaction_display_time, set in the monitor configuration record, or when a new transaction is started.

6.7 Examples of Transaction Info Usage

All important information regarding a transfer is available in the transaction info and accessible in the test environment, and depending on one's needs there are some recommended approaches for how to utilize the transaction info:

1. For complex protocols, where several combinations of data widths and others are possible, it can be complicated for a VVC to determine the structure and constraints of the scoreboard element. Examples of VVCs with such complex protocols are the AXI Stream and the Avalon ST VVCs. Setting up the Scoreboard and checking of received data against expected data for such complex protocols has to be done in the test harness, where the generic scoreboard is instantiated with a constrained scoreboard element and where a dedicated process monitors the VVC transaction info and checks received data with the scoreboard .

2. Transaction info can be used in a DUT Model process that monitors and decodes the actual transaction info data in the test harness. The DUT Model will use the decoded transaction info and add expected data to the VVC scoreboard on the DUT receiving side, e.g. to the SBI_VVC_SB while a SBI VVC is responsible for performing the DUT read access and check received data with SBI_VVC_SB. See section 6.1, example C, for how a DUT Model will appear in the test harness.

6.7.1 Mechanism

The technique of utilizing transaction info in the test environment, be it in a VVC scoreboard support process or in a DUT Model, they all involve the following 3 steps:

1. Wait for the interface trigger signal to be set.
2. Decode the base transaction info operation.
3. Execute wanted operation from the obtained transaction info.

A simple VVC Scoreboard Support process is presented in Figure 5, demonstrating how VVC scoreboard actions can be accomplished using transaction info and a stand-alone process when not performed by a VVC. The same approach is show in Figure 6 with a simple DUT Model process, demonstrating how DUT modelling can be accomplished using transaction info and a stand-alone process. Note that Figure 5 use aliasing to simplify and improve code readability, while Figure 6 use full transaction info names.

6.7.2 Complex Protocols

For scoreboards with complex protocols, e.g. AXI Stream and Avalon ST, the same approach as listed in chapter 6.7.1 applies. The only difference is that the scoreboard element is of a more complex type, i.e. a record element. Figure 7 demonstrate a VVC scoreboard support approach using a complex record element.

Figure 5 VVC Scoreboard Support – with aliasing

```
p_vvc_sb_support : process
  -- transaction info handles
  alias sbi_transaction_trigger     : std_logic is global_sbi_vvc_transaction_trigger(C_SBI_VVC_1);
  alias sbi_transaction_info        : bitvis_vip_sbi.transaction_pkg.t_base_transaction is shared_sbi_vvc_transaction_info(C_SBI_VVC_1).bt;
  alias uart_rx_transaction_trigger : std_logic is global_uart_vvc_transaction_trigger(RX, C_UART_VVC_1);
  alias uart_rx_info                : bitvis_vip_uart.transaction_pkg.t_base_transaction is shared_uart_vvc_transaction_info(RX, C_UART_VVC_1).bt;
  -- helper variable
  variable v_sb_element      : std_logic_vector(C_DATA_WIDTH-1 downto 0);

begin
  while true loop:

    -- Wait for available transaction info
    wait until (sbi_transaction_trigger = '1') or (uart_rx_transaction_trigger = '1');

    if sbi_transaction_trigger'event then
      case sbi_transaction_info.operation is
        when READ =>
          v_sb_element := sbi_transaction_info.data(C_DATA_WIDTH-1 downto 0);
          SBI_VVC_SB.check_received(C_SBI_VVC_1, v_sb_element);
    end if;

    if uart_rx_transaction_trigger'event then
      case uart_rx_info.operation is
        when RECEIVE =>
          v_sb_element := uart_rx_info.data(C_DATA_WIDTH-1 downto 0);
          UART_VVC_SB.check_received(C_UART_VVC_1, v_sb_element);
      end case;
    end if;

  end loop;
end process p_vvc_sb_support;
```

Figure 5 demonstrates the setup of a VVC Scoreboard Support process that operates with the 3 steps listed in Chapter 6.7.1.
For simple scoreboard elements such as std logic vectors these scoreboard approaches are already performed by the VVCs.

Figure 6 DUT Model – no aliasing

```
p_dut_model : process
begin
  while true loop:

    -- Wait for available transaction info
    wait until (global_sbi_vvc_transaction_trigger(C_SBI_VVC_1) = '1') or (global_uart_vvc_transaction_trigger(TX, C_UART_VVC_1) = '1');

    if global_sbi_vvc_transaction_trigger(C_SBI_VVC_1)'event then
      case shared_sbi_vvc_transaction_info(C_SBI_VVC_1).bt.operation is
        when WRITE =>
          UART_VVC_SB.add_expected(shared_sbi_vvc_transaction_info(C_SBI_VVC_1).bt.data(C_DATA_WIDTH-1 downto 0));
    end if;

    if global_uart_vvc_transaction_trigger(TX, C_UART_VVC_1)'event then
      case shared_uart_vvc_transaction_info(TX, C_UART_VVC_1).bt.operation is
        when TRANSMIT =>
          SBI_VVC_SB.add_expected(shared_uart_vvc_transaction(TX, C_UART_VVC_1).bt.data(C_DATA_WIDTH-1 downto 0));
      end case;
    end if;
  end loop;
end process p_dut_model;
```

Figure 6 demonstrates the setup of a DUT Model process that operates with the 3 steps listed in Chapter 6.7.1.
For simple scoreboard elements such as std logic vectors these scoreboard approaches are already performed by the VVCs.

Figure 7 demonstrates the setup of a VVC Scoreboard Support process that operates with the 3 steps listed in Chapter 6.7.1.
For complex scoreboard elements such as records the scoreboard package declaration, defining the shared variable and scoreboard approaches have to be performed outside the VVC.

Figure 7 VVC Scoreboard Support – complex scoreboard element

```
-- define complex Avalon ST scoreboard type
type t_avalon_st_element is record
  channel_value : std_logic_vector(C_CH_WIDTH-1 downto 0);
  data_array    : t_slv_array(0 to C_ARRAY_LENGTH-1)(C_WORD_WIDTH-1 downto 0);
end record t_avalon_st_element;

-- create to_string() function for t_avalon_st_element
function avalon_st_element_to_string(
  constant rec_element : t_avalon_st_element
  ) return string is
begin
  return "channel value: " & to_string(rec_element.channel_value) &
         ", data: " & to_string(rec_element.data_array);
end function avalon_st_element_to_string;

-- define Avalon ST scoreboard
package avalon_st_sb_pkg is new bitvis_vip_scoreboard.generic_sb_package
generic map(t_element         => t_avalon_st_element,
            element_match     => "=",
            to_string_element => avalon_st_element_to_string);
use avalon_st_sb_pkg.all;

shared variable AVALON_ST_VVC_SB : avalon_st_sb_pkg.t_generic_sb;

• • •

p_vvc_sb_support : process
  -- transaction info handles
  alias avalon_st_transaction_trigger : std_logic is
                                        global_avalon_st_vvc_transaction_trigger(C_AVALON_ST_VVC_1);
  alias avalon_st_transaction_info    : bitvis_vip_avalon_st.transaction_pkg.t_base_transaction is
                                        shared_avalon_st_vvc_transaction_info(C_AVALON_ST_VVC_1).bt;
  -- helper variable
  variable v_sb_element               : t_avalon_st_element;
begin
  while true loop:
    -- Wait for available transaction info
    wait until avalon_st_transaction_trigger = '1';

    if avalon_st_transaction_trigger'event then
      case avalon_st_transaction_info.operation is
        when RECEIVE =>
          v_sb_element.channel_value := avalon_st_transaction_info.channel_value(C_CH_WIDTH-1 downto 0);
          v_sb_element.data_array    := avalon_st_transaction_info.data_array(0 to C_ARRAY_LENGTH-1)(C_WORD_WIDTH-1 downto 0);
          AVALON_ST_VVC_SB.check_received(C_AVALON_ST_VVC_1, v_sb_element);
      end case;
    end if;

  end loop;
end process p_vvc_sb_support;
```

7 VVC local sequencers

UVVM testbenches may have one or more central sequencers – also known as test sequencers or test drivers. A single test sequencer is recommended in order to reduce complexity – as synchronization between multiple parallel test sequencer could be really complex.
UVVM does however also provide support for so called local sequencers. These sequencers will typically run inside the VVCs executor process. The executor will typically run a single transaction via a BFM procedure towards the DUT interface, like an sbi_write() or uart_expect() procedure. For more advanced VVCs it would however make sense to send even higher level commands to a VVC, like requesting it to transmit N random bytes, or setting up a peripheral by writing to multiple configuration registers. In these cases, a single command to the VVC will trigger a complete sequence of accesses towards the DUT. The code inside the VVC executors handling these sequences are called local sequencers as they are local to the VVC and thus also improves re-use. These sequences of transactions may also be defined as Compound Transactions (see chapter 6.2).

An example of a local sequencer is the randomisation sequences in the UART VVC, and poll_until in the SBI VVC.

7.1 Local sequencer requirements

The following requirements should be followed when making local sequencers (basically any VVC command resulting in more than one base transaction):

1. If Transaction Info is supported, then both the leaf transaction and the compound transaction info should be updated. (The latter is not required)
2. The sequence should be handled directly inside the VVC executor – and not inside the BFM
   (Otherwise updating the leaf transactions for Transaction Info could be difficult)
3. It should be possible to terminate the sequence immediately after each leaf (or base) transaction – on request from the central sequencer issuing a terminate_current_command() or terminate_all_commands().

8 Protocol aware Error Injection

Error injection into the DUT could be very useful in a testbench in order to test how the DUT handles interface errors when these errors are a) to be detected and corrected, b) detected only, and c) not detected but may or may not affect the behaviour. Protocol aware error injection is defined here as intelligent error injection, given knowledge about the interface and protocol, e.g. to inject a parity error in a protocol rather than just inverting or delaying a signal without pre-defined detailed support to do this at the right place. The latter is supported by a dedicated "brute force" error injection VIP 'bitvis_vip_error_injection' in UVVM.
UVVM has a pre-defined methodology for handling protocol aware error injection in a structured way.
Note that only some VVCs and BFMs currently support error injection. The principles shown for these VVCs and BFMs may be applied directly also for user defined VIP.

8.1 UVVM error injection principles

Error injection may be applied randomly, with no limitations. For UVVM however, we recommend the following approach:

1. No randomisation of behaviour inside BFMs when this could affect the DUT behaviour or output. (and a monitor would be required to check the actual DUT stimuli.)
   Hence BFM procedures should only be called with parameters explicitly defining the interface behaviour (from the BFM side). Thus no parity error randomisation inside.
   The only exception is for behaviour that should not affect the DUT. Thus the position of a data bit error could be randomised inside the BFM.
2. It is recommended that more advanced VVCs include randomisation – in order to distribute this away from the test sequencer and increase the re-use value of a VVC.
   Thus a VVC may be told to apply say 10% parity errors for a UART_VVC transmission into the DUT. In that case the VVC will randomly – with a 10% probability - inject a parity error into the DUT. As the VVC uses a BFM to handle the actual interface/protocol, this means that in 10% of the BFM transmit calls the VVC will request a parity error to be injected.

8.2 Error injection in BFMs

In order to simplify the specification of which errors to inject, the complete error injection specification is given as a sub-record inside the BFM configuration.
E.g. inside the UART BFM configuration the following sub-record is defined – with fields specifying the error injection details (Details given in the UART VIP doc)

- error_injection (fixed name, but type will differ)
  - parity_bit_error (boolean)
  - stop_bit_error (boolean)

In order to initiate error injection, the BFM config record must be modified and included in the BFM procedure call

8.3 Error injection in VVCs

In order to simplify the specification of which errors to inject, the complete error injection specification is given as a sub-record inside the VVC configuration (Note: not the BFM config)
E.g. inside the UART VVC configuration the following sub-record is defined – with fields specifying the error injection details (Details given in the UART VIP doc)

- error_injection
  - parity_bit_error_prob (real between 0.0 and 1.0)
  - stop_bit_error_prob (real between 0.0 and 1.0)

In order to initiate error injection, the VVC config record must be assigned the wanted values via the VVC configuration shared variable. (see ch 4)
Note that the Error injection sub-record inside the VVC configuration will override that of the BFM configuration.
Any compound or more advanced transactions may of course also request error injection directly or indirectly via the VVC command itself.

8.4 Naming and type usage

The error injection sub-record will be VVC and BFM dedicated, and thus any names and types may be used, and even sub-records under 'error_injection' is required. The VVC and BFM error injection records may differ or be the same. The only requirement is that readability is prioritised. Values should be checked against legal ranges or values.

9 Randomisation

UVVM provides functions and procedures for simple generation of random numbers (real, integer, time) and vectors. This is described in detail in the documentation for the UVVM Utility Library. Other libraries for generation of random data may also be used seamlessly with UVVM.
Note that only some VVCs currently include randomisation (e.g. UART TX VVC and SBI VVC.) The principles shown for these VVCs may be applied directly also for user defined VIP.

9.1 UVVM VIP randomisation principles

Randomisation may of course be applied with no limitations in a UVVM based testbench. For UVVM VIP however, we recommend the same general approach as for error injection randomisations:

1. No randomisation of data inside BFMs as this would affect the DUT behaviour or output. (and a monitor would be required to check the actual DUT stimuli.)
   Hence BFM procedures should only be called with explicit data.
2. It is recommended that more advanced VVCs include functionality for randomisation of data – in order to distribute this away from the test sequencer and increase the re-use value of a VVC. Thus, a VVC may be told to apply random data, in which case the VVC will randomly generate data according to a given profile (e.g. uniform) and provide that data to the interface via the BFM call. The profile and constraints will depend on the needs and the VVC implementation

9.2 Data randomisation in BFMs

There is no data randomisation inside a normal BFM, for the reason given above.

9.3 Data randomisation in VVCs

A VVC may be commanded to generate constrained random data, where data in this sense could also be addresses, lengths, etc. Typically such commands would allow flexibility for the number of accesses and other important aspects – like scoreboards, common buffers, files, etc.
A few randomisation profiles have been predefined both as typical use cases and as examples for future extensions, when needed.
The profile names are defined in the type t_randomisation, which is declared in the adaptations package to allow users to add more profiles.

NA	Not applicable (To be used in a record where the field is present, but no randomisation wanted)
RANDOM	Uniform distribution
RANDOM_FAVOUR_EDGES	Significantly more edge cases, where “edge” differs between various interfaces. E.g. UART: Cover patterns like 01111111, 00000000, 11111111, 11111110, 01010101, 10101010,
<user-defined>	

9.4 VVC Command Syntax

See chapter 12 for parameter sequence and options.

10 Testbench Data routing

Transaction Info is providing a mechanism for passively routing source data (data entered into the DUT) out of the VVCs to other parts of the testbench. This data routing is passive in the sense that the transaction data are just provided as a shared variable – for anyone to read. This is covered in chapter 6
There is however also a need for routing data actively inside the testbench, where routing means fetching from or sending to predefined sources and destinations.

10.1 To/from Buffer

UVVM has a global buffer that is divided into multiple smaller buffers that may be indexed and accessed from anywhere in the testbench.
This functionality is described in detail in 'uvm_vvc_framework/doc/UVVM_FIFO_Collection_QuickRef.pdf'.
VVC commands requesting sourcing data from or sending data to these buffers use parameter TO_BUFFER or FROM_BUFFER, followed by the buffer index.

10.2 To scoreboard

Scoreboards may be used anywhere inside the testbench, but for UVVM the following is recommended:

1. Use Scoreboards only on the destination side of the testbench, i.e. where data is received or fetched out of the DUT.
   I.e. for a UART on the DUT UART TX side (= UART_VVC RX side)
2. Every VVC may be connected to one single Scoreboard
3. The Scoreboard instance number should be the same as the VVC instance number
4. When using VVCs make sure the VVC passes the received data to its scoreboard. Do not check the data in the VVC.
   I.e. for a UART_VVC use the receive-command (and not the expect-command) to forward received data to the scoreboard

VVC commands requesting sending data to the scoreboard use parameter TO_SB.

10.3 Data routing options

The profile names are defined in the type t_data_routing, which is defined in UVVM Util types_pkg.

NA	Not applicable (To be used in a record where the field is present, but no data routing wanted)
TO_SB	Data is passed on to the scoreboard for the given VVC
FROM_BUFFER	Data is sourced from the UVVM global buffer
TO_BUFFER	Data is also sent to the UVVM global buffer
TO_FILE	TBD – Not yet implemented (Do not use – as this may change)
FROM_FILE	TBD – Not yet implemented (Do not use – as this may change)
<user-defined>	

10.4 VVC Command Syntax

See chapter 12 for parameter sequence and options.

11 Controlling property checkers

A major VVC advantage is that lots of additional very useful functionality may be added inside the VVC entity, meaning that all the verification support for a given interface can be encapsulated inside a single VHDL entity. A major advantage of UVVM is that adding additional functionality and controlling it from the test sequencers is really simple. One very useful additional functionality is property checkers, and some typical examples of this could be to check the minimum allowed bit period, the minimum inter-packet gap, back-to-back restrictions, etc., or in general to check a given requirement continuously – especially when this is easier to do outside the BFM – for instance in a dedicated checker process.

A dedicated checker process would typically just wait for a trigger condition on the interface (like a UART data bit changing its level), then wait again for a next trigger (the next data bit), and then check that the time between the two changes is not less than the minimum allowed bit period. This check could then be repeated forever. It is however recommended that the check could be turned on and off for more flexibility.

11.1 Property check configuration

In UVVM turning checkers on and off is controlled by the VVC configuration (chapter 4), and often additional control of the checker behaviour is also required. Thus, it is recommended to include the checker control for each individual check in a dedicated sub-record. An example on this (for the UART VVC) is shown below. See UART_VVC RX for implementation.

```
.bit_rate_checker                       -- Sub-record containing all control of the property checker behaviour
    .enable         boolean             -- Enables or disables the complete bit rate checker.
    .min_period     time
    .alert_level    t_alert_level
```

For this example, the bit rate checker inside the UART_VVC RX will trigger on changes on the DUT TX and execute the check if enable is TRUE.

11.2 Setting up the configuration

The bit rate checker configuration may be changed directly from the sequencer via the shared variable VVC configuration.

12 VVC parameters and sequence for Randomisation, Sources and Destinations

In order to assure a common syntax and understanding for the various VVC commands controlling these features, the sequence and type of parameters have been defined as follows:

Parameter sequence	Preceding command part	[Number of repetitions]	Randomness	Data routing type	[Data routing index]
Example a	uart_transmit(UART_VVCT,1,TX,	4	RANDOM_FAVOUR_EDGES	TO_BUFFER	5
Example b	uart_receive(UART_VVCT,1,RX,			TO_SB	

Example a means: make 4 transactions with random data (using predefined profile RANDOM_FAVOUR_EDGES) and send the data also to BUFFER 5
(e.g. uart_transmit(UART_VVCT,1,TX, 4, RANDOM_FAVOUR_EDGES, TO_BUFFER, C_UART_BUFFER, “my message”);
Example b means: keep on receiving data and send the received data also to the local Scoreboard

Exactly what variants will be available for each VVC is up to the VVC designer, but this gives the sequence and the options.

13 Multiple Central Sequencers

A structured test environment is important, and we recommend the use of a structured test harness to instantiate VVCs, DUT, clock generator and so forth. The testbench may consist of one or more test sequencers which are used to control the complete testbench architecture with any number of VVCs, although for a better testbench overview we recommend having a single central test sequencer only – for most testbenches.

14 Monitors

Monitors could be great to check the interface accesses to a DUT – to report the transaction to the testbench – with all relevant info like operation (write, read, transmit, …), data, address, etc. This information may be critical in order to understand the operation of the DUT and its expected outputs. A monitor is not a protocol checker, but may of course check various properties of an interface/protocol. A typical Monitor will however only provide the relevant basic information and leave more advance interpretation to other parts of a testbench. For simple protocols like the UART, UVVM also includes basic error checking in the Monitor – as this happens at a very low level. For more advanced protocols it would make sense to just pass on the low level info to a higher level checker. The reason for making a dedicated monitor rather than leaving that to the testbench model is to achieve a better testbench structure and more efficient reuse.

It should however be mentioned that implementing Transaction Info (ch. 6) inside a VVC significantly reduces the need for a dedicated monitor, as the VVC will then be able to pass the complete transaction information on to for instance a model inside the Testbench.

14.1 Transfer of Monitor information to the testbench

The mechanism for passing the monitor deduced transaction out of the monitor is almost exactly the same as for passing transaction info out of a VVC – as described in chapter 6. The only difference is that the monitor can only provide parts of what the VVC can provide – as follows compared to the tables given in chapter 6.

- No monitor can provide info about compound transactions
- For a split transaction protocol like Avalon – only the sub-transactions could be provided (which could be analysed at the higher level to provide Base transactions)
- A monitor cannot provide meta data like command index or command message.

As the monitor does not know what to expect at the beginning of a transaction the following field limitations apply:

- Operation: Can only be known some time after the start of the transaction. Will be set when the type of the transaction is known, e.g. TRANSMIT or RECEIVE for UART (otherwise NO_OPERATION)
- Transaction_status: Will be set to FAILED or SUCCEEDED as soon as the result is 100% given. Prior to that – during the transaction : IN_PROGRESS. FAILED/SUCCEEDED will remain for the transaction_display_time given inside the monitor configuration record, or until the next transaction FAILED or SUCCEEDED.

An example of a complete monitor is shown in the UART VIP directory.

14.2 Transaction info transfer signals

The Transaction info provided out of a Monitor uses a set of a global signal and a shared variable. These and all related VHDL types are defined in transaction_pkg.
- Monitor Transaction Info trigger signal : global_<protocol-name>_monitor_transaction_trigger, e.g. global_uart_monitor_transaction_trigger(channel, instance number)
- Monitor Transaction Info shared variable : shared_<protocol-name>_monitor_transaction_info, e.g. shared_uart_monitor_transaction_info(channel, instance number)

See Transaction Info record hierarchy in Table 5 for more details.

15 Compile scripts

In the script folder in the root directory the compile_all.do compiles all UVVM components. This script may be called with one to three input arguments. The first input argument is the directory of the script folder at the root directory from the working directory. The second input argument is the target directory of the compiled libraries, by default every library is compiled in a sim folder in the corresponding components directory. The third input argument is the directory to a custom component list in .txt format. The script will only compile the components listed in that file. By default, the script uses the file component_list.txt located in uvvm/script. This file can be modified so that only some components are compiled.

Example: do uvvm/script/compile_all.do uvvm/script

There are also compile scripts for all UVVM components located in the script folder of each UVVM component. These scripts can be called with two input arguments. The first input argument is the directory of the component folder from the working directory. The second input argument is the target directory of the compiled library, default is the sim folder in the respective component.

Example: do uvvm/uvvm_vvc_framework/script/compile_src.do uvvm/uvvm_vvc_framework

16 Scope of verbosity control

Message IDs are used for verbosity control in many of the procedures and functions in UVVM, as well as log messages and checks in VVCs, BFMs and Scoreboards.
Note that VVCs and Scoreboards comes with dedicated message ID panels and are not affected by the global message ID panel, but accessed by addressing targeting VVC or Scoreboard and, if applicable, instance number or with a broadcast.
However, when a VVC is executing commands triggered by an HVVC (Hierarchical-VVC), e.g. SBI write due to Ethernet transmit, the VVC will use the HVVC's ID panel instead. See the Hierarchical VVCs section in this document for an example of the HVVC structure.

E.g.

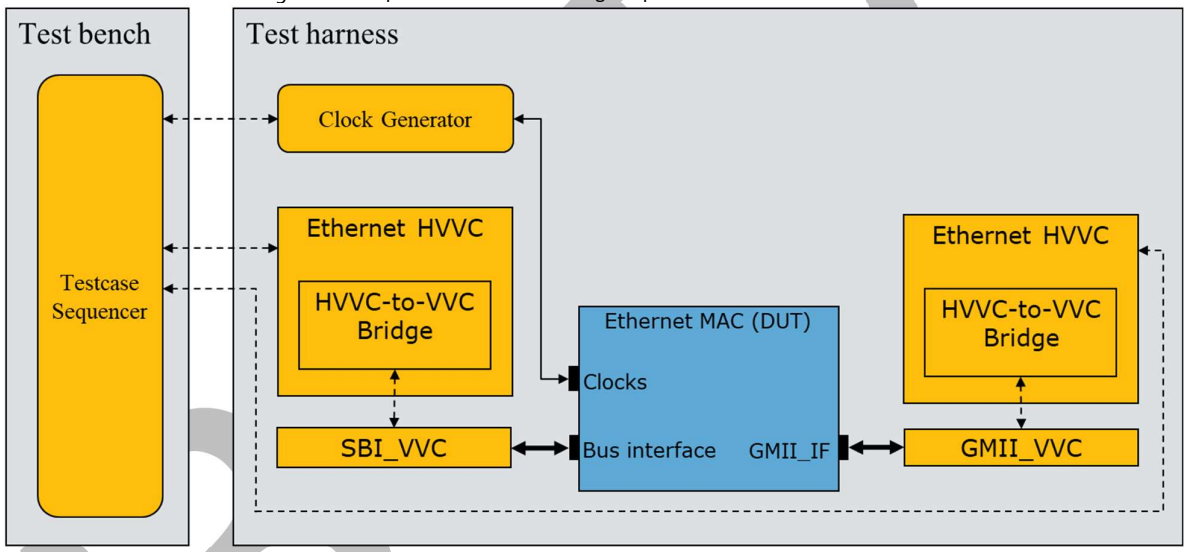
```
-- Global message ID panel. Does not apply to VVCs or Scoreboards, as they have their own local message ID panel
disable_log_msg(ALL_MESSAGES);
enable_log_msg(ID_SEQUENCER);

-- VVC message ID panel
disable_log_msg(VVC_BROADCAST, ALL_MESSAGES); -- broadcast to all VVCs and instances
enable_ log_msg(I2C_VVCT, C_VVC_INSTANCE_1, ID_BFM_WAIT); -- I2C VVC instance 1
enable_log_msg(I2C_VVTC, C_VVC_INSTANCE_2, ID_BFM_WAIT); -- I2C VVC instance 2

-- Scoreboard message ID panel
shared variable sb_under_test : record_sb_pkg.t_generic_sb;
…
sb_under_test.disable_log_msg(ALL_INSTANCES, ID_CTRL); -- broadcast to all SB instances
sb_under_test.enable_log_msg(C_SB_INSTANCE_1, ID_DATA); -- SB instance 1
```

A subset of message IDs is listed in UVVM Utility Library QR, section 1.10.

17 Hierarchical VVCs

Many protocols and applications consist of several abstraction levels, e.g. physical layer, link layer, transaction layer, etc. When writing a test case for a higher level you most likely want to ignore the underlaying levels and only deal with the scope of the relevant level. The test case will be less complex and easier to both write and read. A hierarchical VVC (HVVC) is a VVC of a higher protocol level than the physical layer, i.e. it has no physical connections. The test case only communicates with the HVVC which communicate with the lower level. Data is propagated upwards and downwards between the HVVC and DUT through a standard VVC connected to the DUT.
The HVVC-to-VVC Bridge is the connection between a hierarchical VVC (HVVC) and the VVC at a lower protocol level, in this context referred to only as the VVC. Communications between the HVVC and VVC is handled by the HVVC-to-VVC Bridge. Data is transferred between the HVVC and HVVC-to-VVC Bridge on a common interface and converted in the HVVC-to-VVC Bridge to/from the specific interface of the VVC used. An example of this concept used on Ethernet is seen in Figure 8.

Figure 8 Example of HVVC-to-VVC Bridge implemented in an Ethernet HVVC.



17.1 HVVC usage

To simulate an HVVC you only need to do the following:

1. Instantiate the HVVC in the test harness and set the generic GC_PHY_INTERFACE to the physical interface you want to use.
2. Instantiate the VVC of the physical interface with the same instance index as GC_PHY_VVC_INSTANCE_IDX.
3. Connect the VVC of the physical interface to the DUT.

E.g.

```
i1_ethernet_vvc : entity bitvis_vip_ethernet.ethernet_vvc
  generic map(
    GC_INSTANCE_IDX         => C_VVC_ETH,
    GC_PHY_INTERFACE        => GMII,
    GC_PHY_VVC_INSTANCE_IDX => C_VVC_GMII
  );

i1_gmii_vvc : entity bitvis_vip_gmii.gmii_vvc
  generic map (
    GC_INSTANCE_IDX => C_VVC_GMII
  )
  port map (
    gmii_vvc_tx_if => gmii_vvc_tx_if,
    gmii_vvc_rx_if => gmii_vvc_rx_if
  );
```

Any VVC can be used as a physical interface, however it needs to have an HVVC-to-VVC Bridge implementation. You can find the available implementations under bitvis_vip_hvvc_to_vvc_bridge/src.
For information on how to implement your own, see HVVC_to_VVC_Bridge_Implementation_Guide located in bitvis_vip_hvvc_to_vvc_bridge/doc.