VVC Implementation Guide

This guide is meant for users that want to make their own VVC.
Users that only write test cases that are using the VVCs do NOT need to read this guide.

Making test cases using available VVCs is very easy.
Implementing new VVCs is slightly more complex, but fast, safe and efficient once you understand the VVC mechanisms.

Understanding and Modifying a VVC

This guide goes through all relevant files needed to make a complete VVC.
The intention is to allow a VVC implementer to go through file by file and understand and modify as needed.
All code objects and functionality given in the VVC and UVVM files are considered mandatory unless otherwise noted.

To implement your own VVC

Prerequisites:

1. Go through '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.
2. Read the 'VVC_Framework_manual.pdf' to understand the basic concepts, the communication and handshake between the central sequencer and the VVCs.
3. If your VVC is going to access a DUT interface, you need to have a BFM (Bus Functional Model) for that interface – independent of whether you are going to make a VVC.

Implementing your new VVC:

1. Use the script *vvc_generator* located in the uvvm_vvc_framework/script/vvc_generator/ to generate a new VVC. Notice that the name length is limited by C_LOG_SCOPE_WIDTH (default =20) in uvvm_util/src/adaptations_pkg.
2. Then see comments in the code for where to make required changes.
3. If the new VVC uses multiple channels other than TX and RX, modify the t_channel type under UVVM-Util adaptations_pkg.
4. See this guide for an explanation to all the various sections you need to evaluate or modify – file by file.

UVVM™

1 Dependent and independent source

One of the key concepts of the UVVM VVC Framework is the compilation strategy, and how some packages in the UVVM VVC Framework directory are compiled into each of the individual VVC libraries. To avoid confusion about this for future VVC designers, the VVC dependent and VVC independent sources have been marked and split into two source directories. The target dependent source, also known as packages that are compiled into each of the individual VVCs, are placed in the *src_target_dependent/* folder. These files are also prefixed with "*td_*"* for "target dependent".
The target independent files are compiled into the *uvvm_vvc_framework* library. These files are placed in the *src/* folder and prefixed with *"ti_*"* for "target independent".

For examples of how the compile order should be handled, please see the example VIPs' QuickRef and Modelsim compile scripts.

2 <name>_vvc.vhd

2.1 For single channel VVCs

Code Section	Description
Entity	GC_INSTANCE_IDX: Needed in case there are multiple instances of a given VVC. (E.g. DUT with 2 <VVC-NAME>S). Default is 1, but any natural type is ok. GC_<name>_CONFIG: Recommended. Allows predefined BFM behaviour to be set up for every VVC. GC_CMD_QUEUE_*: Needed to limit the queue size and to generate a warning if more elements in the queue than ever expected. Other generics: Optional and VVC dependent. These generics could for example contain widths of BFM signals. The interface to the DUT and any other needed I/O. The examples show DUT interfaces as single signals, records and records of records. This is optional.
Declarations	C_SCOPE: Used for logs and alerts. C_VVC_NAME is defined in the VVC 'vvc_methods_pkg.vhd' C_VVC_LABELS: A record of constants, e.g. name and channel, used in multiple procedures. Various status signals used as flags between the processes Command termination record. (Fields: *set*, *reset*, *is_active*. Where *set* and *reset* signal fields are used to toggle *is_active*. Used as inter process flags. *command_queue* is the queue of commands to be executed in sequence towards the DUT. The aliases are defined to allow common and simplified names.
Constructor	The constructor is run once only – immediately when starting the simulation. The procedure: - Initialises VVC with BFM config and the queue with queue name - Allows constructor log for VVC info (using ID_CONSTRUCTOR), and VVC Queue info (using ID_CONSTRUCTOR_SUB) The procedure will report alerts with severity TB_FAILURE if one of the following occurs: - The instantiated VVC index is higher or equal to the maximum allowed number of VVC instances, given by C_MAX_VVC_INSTANCE_NUM in UVVM-Util 'adaptations_pkg.vhd' - UVVM has not been initialized

Command Interpreter

Waits for commands from the central test sequencer distributed to this VVC, then puts the command on the queue for execution or immediately performs the required action – depending on command type. Afterwards, it acknowledges the command and waits for the next command from the sequencer.

Step 0
Initialize_interpreter():
- Initialises parameters to default passive/initial values (e.g. terminate_current_cmd.set := '0')

Step 1
await_cmd_from_sequencer():
- Waits for a command from the central sequencer. Continues on matching VVC, instance index, name and channel
- Log at start using ID_CMD_INTERPRETER_WAIT and at the end using ID_CMD_INTERPRETER.
- Will only accept exact matches of instance index and name, and either the correct address or "ALL_CHANNELS"

Step 2a (Only if command type is QUEUED)
put_command_on_queue():
- Puts the received command on the VVC queue (for later retrieval by the Command Executor)

Step 2b (Only if command type is IMMEDIATE)
Execute the requested command/operation.
- For the VVC methods these procedures will correspond to the UVVM methods, but prepended with "interpreter_", e.g. "interpreter_await_completion". These UVVM methods are documented in the Common_VVC_methods.pdf document. Other commands are documented in their respective QuickRefs.
- *format_commmand_idx()*: (ti_vvc_framework_support_pkg) Converts the command index to string, enclosed by C_CMD_IDX_PREFIX and C_CMD_IDX_SUFFIX (found in UVVM-Util adaptations_pkg).

Step 3
acknowledge_cmd():
- Acknowledges the command from the sequencer by driving global_vvc_ack signal (to '1') for 1 delta cycle, then setting it back to 'Z'. This lets the central sequencer know that it can continue execution.

Command Executor

Fetches commands from the command queue – if any. Then executes the command and fetches or waits for the next command in the command queue

Step 0
initialize_executor():
- Initialises parameters to default passive/initial values (e.g. terminate_current_cmd.reset := '0')

Step 1
fetch_command_and_prepare_executor():
- Fetches a command from the queue (waits until available if needed).
- Sets relevant flag parameters.
- Log command using ID_CMD_EXECUTOR (or Log using ID_CMD_EXECUTOR_WAIT if queue is empty)

Set transaction information for wave-view.
- transaction_info_for_waveview (from vvc_methods_pkg.vhd) is a shared variable intended for use in a wave-view – to yield a better overview of transaction info. Setting this information is optional. The pad_string and to_string procedures are documented in the UVVM-Util QuickRef.

Insert inter BFM delay if requested.
- insert_inter_bfm_delay_if_requested(): Inserts either start-to-start or finish-to-start delay between BFM accesses if this is set in the inter_bfm_delay parameter in 'vvc_config'. Logs information using ID_CMD_EXECUTOR.
- If the command currently being processed by the executor is a BFM access, a timestamp will be stored in *v_timestamp_start_of_current_bfm_access*.

Step 2

	Executes a command depending on the requested command/operation. - *terminate_current_cmd* is only checked inside operations that require multiple BFM accesses – like for instance a POLL_UNTIL command. - *store_result()* is executed for any BFM, where it makes sense for you to store the result of a BFM access. In our example for SBI we think it only makes sense for 'READ'. - Logging as defined by your BFM. - Transaction info can be stored in the *transaction_info_for_waveview* struct for each command type, but this is optional. Update the BFM access timestamps if this was a BFM access. - *v_timestamp_of_last_bfm_access* is set to *now* - *v_timestamp_start_of_last_bfm_access* is set to *v_timestamp_start_of_current_bfm_access* The *terminate_current_cmd* flag is reset if it has been active. Update the *last_cmd_idx_executed* variable with the current command index, *v_cmd.cmd_idx*.
Command Terminator	The command terminator concurrent procedure sets the *is_active* flag based on the *set* and *reset* flags.

2.2 Additional for multi-channel VVCs

Please note that we strongly recommend implementing the VVCs such that each leaf VVC handles one independent DUT communication thread (here: 'Channel'). No more; no less. This allows a single command queue and a single executor handling DUT communication. (Additional processes to handle other characteristics is fine. E.g., a parallel bit-rate check thread.)
Note that SBI_VVC must handle both read and write accesses, but never simultaneously and always in the given order.

Multi-channel VVCs may be implemented in many different ways – depending on your preferences and priorities. Some examples:

1. **As unique VVC implementation**
   Unique VVCs may be used in order to omit the channel input, e.g. UART RX VVC and UART TX VVC. UART TX VVC would only contain TX specific BFM procedures, while UART RX VVC would only contain RX specific BFM procedures. With this approach the test bench sequencer calls would look like e.g. (assuming both VVCs in this pair are set to instance index 1):
   a. *uart_transmit(UART_TX_VVCT,1,…)*
   b. *uart_receive(UART_RX_VVCT,1,…)*
2. **As shared VVC implementation with usage restricted by user, and multiple VVC instances**
   A combined VVC with different VVC instances for different channels e.g. RX and TX. The TX instance could e.g. be instance 1, and the RX instance could be e.g. instance 2. Using this UART VVC with this implementation would look like:
   a. *uart_transmit(UART_VVCT,1,…)*
   b. *uart_receive(UART_VVCT,2,…)*
3. **As shared VVC implementation with GC_CHANNEL generic input**
   A combined VVC with the same combined VVC implementation, but separate instances for different channels e.g. RX and TX (both functionalities inside the same leaf VVC). The downside of this implementation is that it would be possible to call TX BFM procedures when calling the RX VVC channel. Using this UART VVC would look like:
   a. *uart_transmit(UART_VVCT,TX,1,…)*
   b. *uart_receive(UART_VVCT,RX,1,…)*
4. **As unique VVC implementation with GC_CHANNEL generic input**

This approach uses unique VVC implementations for each channel, e.g. in uart_rx_vvc.vhd and uart_tx_vvc.vhd, but they both share the VVC target parameter, UART_VVCT. They both use the GC_CHANNEL generic input to specify their channel, i.e. TX or RX. This is similar to the method described in 3., but with restrictions that ensure that e.g. the UART TX VVC can't use the UART RX BFM procedures. The included bitvis_vip_uart example is implemented with this method. Using this UART VVC would look like:

a. *uart_transmit(UART_VVCT,TX,1,…)*
b. *uart_receive(UART_VVCT,RX,1,…)*

When using multiple leaf VVCs it is recommended to use a wrapper architecture to encapsulate the channels. This way, it is possible to instantiate a single VVC rather than each VVC channel individually. For more information about the wrapper architecture, see the uart_vvc.vhd example in the bitvis_vip_uart/src/ directory.

3 vvc_cmd_pkg.vhd

Section	Comment
t_operation	Contains all UVVM common operations, e.g. AWAIT_COMPLETION and ENABLE_LOG_MSG, in addition to the VVC specific operations such as e.g. WRITE and READ. The VVC specific will have to be evaluated and potentially replaced when implementing a new VVC. The t_operation type is used when relaying commands from the sequencer to the VVC. The t_operation type also has its own to_string() function in this package.
t_vvc_cmd_record	Record type used for relaying a command from the test bench sequencer to the VVC. The record contains fields needed in the common UVVM procedures (listed under the "Common UVVM fields" comment), and VVC specific fields needed to relay data to the VVC executor. The VVC specific data fields should contain any data fields that the BFM procedures might need, e.g. data, address, timeouts etc. There is also a default for this type called C_VVC_CMD_DEFAULT in this package.
Constants	The vvc_cmd_pkg.vhd should contain constants that needs to be set for the entire VVC. In most VVCs this will include the C_VVC_CMD_STRING_MAX_LENGTH which determines the maximum size of msg variables in the VVC. It is also a good idea to declare constants for maximum VVC data/address bus sizes here. It will be possible to construct VVCs with bus sizes up to and including the sizes declared here.
Shared Variables	The shared_vvc_cmd shared variable (type t_vvc_cmd_record) is used for relaying commands between sequencer methods and the VVC. It is default set to C_VVC_CMD_DEFAULT, which is also declared in this file.

4 vvc_methods_pkg.vhd

Section	Comment
Constants and aliases	The vvc_methods_pkg contain constants for the VVC name, e.g. “<NAME>_VVC”. There are also aliases created to make the code more readable.
<NAME>_VVCT	The <NAME>_VVCT signal (e.g. SBI_VVCT) is the VVC target record signal. The target type *t_vvc_target_record* is a record that contains the parameters needed to trigger a VVC, and to identify the correct target of a VVC command.
t_vvc_config	This type contains the needed configuration for setting up the VVC and BFM. In Bitvis VVCs the BFM configuration is encapsulated in a bfm_config record, of type t_<bfm_name>_bfm_config. This record is placed in this file and compiled into each VVC since the VVC/BFM configuration will differ for each VVC. Record contents: - inter_bfm_delay: A record containing the potential inter-bfm delay specifications, e.g. if BFM accesses shall be separated with a given time - cmd_queue_count_*: Command queue specifications - msg_id_panel: The ID panel that the VVC shall use - bfm_config: A record containing all settings for the BFM, e.g. clock periods, message IDs etc. A constant *C_<name>_VVC_CONFIG_DEFAULT* is defined for this type to use as default value. A shared variable array of t_vvc_config *shared_<name>_vvc_config* is declared and all elements are set to the default value.
t_vvc_status	The optional status record is created in order for the test bench sequencer to have access to the status of the VVC. The status record can contain anything that is relevant for the outside, and it is recommended to have at least these three fields: - current_cmd_idx: The current command index being processed in the executor - previous_cmd_idx: The previous command index being processed in the executor - pending_cmd_idx: The number of pending commands to be processed by the executor A constant *C_VVC_STATUS_DEFAULT* is defined for this type to use as default value. A shared variable array of t_vvc_status *shared_<name>_vvc_status* is declared and all elements are set to the default value.
t_transaction_info_for_waveview	The t_transaction_info_for_waveview type is an optional status record to be used in the wave-view. This record should be modified to suit the BFM fields, but it can also contain the VVC field t_operation, which can be updated with the VVC operation currently being processed by the executor. The transaction_info_for_waveview is meant as a way of improving the readability of wave-views. A constant *C_TRANSACTION_INFO_FOR_WAVEVIEW_DEFAULT* is defined for this type to use as default value. A shared variable of t_transaction_info_for_waveview *t_<name>_transaction_info_for_waveview* is declared and all elements are set to the default value.
VVC Dedicated Methods	The vvc_methods_pkg.vhd file also contains the VVC procedures that are called from the test bench sequencer. These procedures should reflect the procedures in the BFM, e.g. <name>_write or <name>_receive. The parameters of these procedures are mostly up to the user, but it is recommended that the BFM arguments that are rarely altered be placed into the bfm_config parameter, while the parameters that changes often are used as input arguments. Since these VVC methods are reused for all instances of this VVC, it is necessary with some extra parameters in order to specify which VVC instance to forward the call to. This is done with the first two (or three) parameters:

- *signal VVCT : inout t_vvc_target_record;*
- *constant vvc_instance_idx : in integer;*
- *constant channel : in t_channel; -- Only if the VVC is multi-channel.*

The method bodies are quite similar for all VVC commands:

1. First, the shared_vvc_cmd record is set to its default value, resetting the data from any potential previous command.
2. The general VVC fields (e.g. name and instance index) are set using the UVVM method set_general_target_and_command_fields()
3. The VVC specific fields are set in the shared_vvc_cmd shared variable. This means e.g. address and data fields.
4. The command is sent to all VVCs using the UVVM method *send_command_to_vvc(VVCT)*

All VVC instances and channels of this type receive the command, but only the VVC with the correct instance index, channel and name will accept it and acknowledge it.

5 BFM prerequisites

There are no firm restrictions of how to implement the BFM in order for the VVC to function, but if the VVC generated with the *vvc_generator* script is to work out of the box, it is necessary to have some components in the BFM:

- The BFM needs to be called <name>_bfm_pkg.vhd. If this is not the case, the package use clauses in each of the VVC files needs to be altered.
- The BFM needs to contain a bfm_config record type with an associated default constant. The generated VVC file assumes that this bfm config type is called *t_<name>_bfm_config* and the constant is called *C_<NAME>_BFM_CONFIG_DEFAULT*. In order to support the delay operation in the VVC executor the BFM config type will also need to have a parameter *clock_period*. If this is not needed, the "INSERT_DELAY" case in the generated VVC can be removed.

A BFM skeleton that contains the necessary structure is created by the *vvc_generator* script, and can be used as a base for a BFM that includes the necessary structure for the VVC to work out of the box.

6 UVVM Framework Packages

6.1 td_target_support_pkg.vhd

The UVVM VVC dedicated support package contains VVC support that is common for all VVCs, but needs to be compiled specifically into each of the VVC libraries.

Section	Comment
Target record	The target record type, *t_vvc_target_record*, is used to target a VVC command to a specific VVC implementation. This is needed since many of the UVVM common commands are shared between all VVCs, e.g. await_completion() which is compiled into each VVC library. For a sequencer with two VVCs, A and B, there must be a way of determining if await_completion is to be executed in VVC A or VVC B. To resolve this, each VVC has a signal in their vvc_methods_pkg that is compiled into their own library. For VVC A and B this signal will be called A_VVCT and B_VVCT. When await_completion(A_VVCT,…) is called from the sequencer, the compiler will understand that this await_completion is called with target type library_a.t_vvc_target_record, which only complies with the await_completion procedure in the VVC A library . td_target_support_pkg also contains a default value for the t_vvc_target_record type, and a function *set_vvc_target_defaults* for setting the VVC target based on the VVC name.
String methods	The package contains two string methods: - to_string(): This function converts a t_vvc_target_record, vvc_instance and vvc_channel into a string - format_command_idx(): Function which encapsulates a command record index.
send_command_to_vvc	Sends command to VVC and waits for ACK or timeout - Logs with ID_UVVM_SEND_CMD when sending to VVC - Logs with ID_UVVM_CMD_ACK when ACK or timeout occurs
Setting the command field	Sets target index and channel, and updates shared_vvc_cmd which is used to transport VVC commands from the central test bench sequencer to VVC.

6.2 td_vvc_entity_support_pkg.vhd

The VVC support package contains procedures that are compiled into and used in the VVC. This includes initializers for the executor and interpreter, and the interpreter procedures called *interpreter_**, e.g. *interpreter_await_completion*. For more information about the *interpreter_** procedures, please see the Common_VVC_Methods under doc/. For more information about the other methods in this package, see the <name>_vvc.vhd section in this document.
In addition to the procedures, the td_vvc_entity_support_pkg also contains types for VVC labels and executor results. The result array is also defined and its shared variable is instantiated in this package.

7 Additional Documentation

Additional documentation about UVVM and its features can be found under “uvvm_vvc_framework/doc/”.