UVVM
Bringing UVM to VHDL

DVCon U.S. 2022, 3rd March 2022
Independent Design Centre for Embedded Systems and FPGA

Established 1\textsuperscript{st} of January 2021. Extreme ramp up

- January 2021: 1 person
- January 2022: → 19 persons (SW:6, HW:2, FPGA:10) - And still growing...

Continues the legacy from bitvis

- All previous Bitvis technical managers are now in EmLogic

- Verification IP and Methodology provider UVVM

- Course provider within FPGA Design and Verification

  - Accelerating FPGA Design (Architecture, Clocking, Timing, Coding, Quality, Design for Reuse, ...)
  - Advanced VHDL Verification – Made simple (Modern efficient verification using UVVM)
What is UVVM?

UVVM = Universal VHDL Verification Methodology

- **VHDL** Verification Library & Methodology
- Free and Open Source
- Very structured infrastructure and architecture
- Significantly improves Verification Efficiency
- Assures a far better Design Quality
- Recommended by Doulos for Testbench architecture
- ESA projects to extend the functionality
- IEEE Standards Association Open source project
- Included with various simulators
- Runs on GHDL
VHDL was declared as "dying" already in 2003
- and in 2007, - and ....

BUT - According to Wilson research September 2020:

**VHDL for FPGA: >50% world-wide**

- For VHDL designers
- Low user threshold
- Logical evolution on VHDL
- Low cost solution
- More efficient VHDL designers
UVVM – World-wide

- Number 1 world-wide for VHDL verification *1
- Number 1 in Europe, indep. of language *1
- Number 2 world-wide, indep. of language *1
- By far, the fastest growing, indep. of language *1

* According to Wilson Research, per Sept. 2020

** Multiple answers possible
Example on test sequencer code and transcript/log

clock_generator(clk, GC_CLK_PERIOD);

log(ID_LOG_HDR, "Check Interrupt trigger clear mechanism");
check_value(irq2cpu, '0', "irq2cpu default inactive");
check_stable(irq2cpu, now - v_reset_time, "Stable irq2cpu");
gen_pulse(irq_source, '1', clk_period, "Set IRQ source for clock period");
await_value(irq2cpu, '1', 0 ns, 2* C_CLK_PERIOD, "Interrupt expected");
sbi_write(C_ADDR_ITR, x"AA", "ITR : Set interrupts");
Result in Modelsim
Testbench infrastructure library

- `log()`, `alert()`, `report_alert_counters()`
- `check_value()`, `await_value()`
- `check_stable()`, `await_stable()`
- `clock_generator()`, `adjustable_clock_generator()`
- `random()`, `randomize()`
- `gen_pulse()`
- `block_flag()`, `unblock_flag()`, `await_unblock_flag()`
- `await_barrier()`
- `enable_log_msg()`, `disable_log_msg()`
- `to_string()`, `fill_string()`, `to_upper()`, `replace()`, etc...
- `normalize_and_check()`
- `set_log_file_name()`, `set_alert_file_name()`
- `wait_until_given_time_after_rising_edge()`
- etc...
Simple data communication

May use Utility Library and provided BFM's

Free, Open source BFM's:
- UART, AXI4-lite, SPI, I2C
- Avalon MM, AXI4-stream
- Avalon Stream, GPIO, SBI, GMII, RGMII, ...

Quick References are provided

---

**TB**: 172 ns. `uart_tb` `uart_transmit(x"2A")` on UART RX

**TB**: 192 ns. `uart_tb` `sbi_check(C_RX, x"2A")` completed. From UART RX

**TB**: 192 ns. `uart_tb` `sbi_write(C_TX, x"B3")` completed. To UART TX

**TB**: ERROR:

**TB**: 192 ns. `uart_tb`

**TB**: value was: 'xB2'. expected 'xB3'.

**TB**: (From `uart_expect(x"B3")`)

---

**p_main (test-sequencer)**

- `uart_transmit(x"2A")`
- `sbi_check(C_RX, x"2A")`
- `sbi_write(C_TX, x"B3")`
- `uart_expect(x"B3")`
AXI-stream - BFM based TB
- as simple as possible

- No test harness (for simplicity)
- Sequencer has direct access to DUT signals
  - Thus BFMs from p_main can also see the DUT signals

- Simplified UVVM
  - For simple usage
  - Subset of UVVM
  - No VVCs or VCC support
  - All BFMs in the same directory and library

Only need to download from Github (clone or zip) and compile (total 5 min)
Resulting transcript + Debug

Note: Removed Prefix and Scope to show on a single line.

```vhdl
axistream_transmit(v_byte_array, msg, clk, m_axis);
```

<table>
<thead>
<tr>
<th>ID_BFM</th>
<th>106.0 ns</th>
<th>axistream_transmit(3B) =&gt; Tx DONE.</th>
</tr>
</thead>
</table>

```vhdl
axistream_expect(v_exp_array(0 to 2), "", clk, s_axis);
```

<table>
<thead>
<tr>
<th>ID_BFM</th>
<th>122.0 ns</th>
<th>axistream_expect(3B) =&gt; OK, received 3B.</th>
</tr>
</thead>
</table>

May add more info for debugging

```vhdl
enable_log_msg(ID_PACKET_INITIATE); enable_log_msg(ID_PACKET_DATA);
```

<table>
<thead>
<tr>
<th>ID_PACKET_INITIATE</th>
<th>52.0 ns</th>
<th>axistream_transmit(3B) =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_PACKET_DATA</td>
<td>52.0 ns</td>
<td>axistream_transmit(3B) =&gt; Tx x&quot;00&quot;, byte# 0.</td>
</tr>
<tr>
<td>ID_PACKET_DATA</td>
<td>68.0 ns</td>
<td>axistream_transmit(3B) =&gt; Tx x&quot;01&quot;, byte# 1.</td>
</tr>
<tr>
<td>ID_PACKET_DATA</td>
<td>82.0 ns</td>
<td>axistream_transmit(3B) =&gt; Tx x&quot;02&quot;, byte# 2.</td>
</tr>
<tr>
<td>ID_PACKET_COMPLETE</td>
<td>106.0 ns</td>
<td>axistream_transmit(3B) =&gt; Tx DONE.</td>
</tr>
</tbody>
</table>

May add similar debugging info for data reception
Documentation BFM

Similar docs for all BFM

UVVM - Bringing UVM to VHDL
# AXI4-Stream BFM – Quick Reference

**Configuration**
- Protocol Behaviour
- Compliance checking
- Simulation set-up

**BFM Configuration record** `t_axistream_bfm_config`

<table>
<thead>
<tr>
<th>Record element</th>
<th>Type</th>
<th>C_AXISTREAM_BFM_CONFIG_DEFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_wait_cycles</td>
<td>integer</td>
<td>100</td>
</tr>
<tr>
<td>max_wait_cycles_severity</td>
<td><code>t_alert_level</code></td>
<td>ERROR</td>
</tr>
<tr>
<td>clock_period</td>
<td>time</td>
<td>-1 ns</td>
</tr>
<tr>
<td>clock_period_margin</td>
<td>time</td>
<td>0 ns</td>
</tr>
<tr>
<td>clock_margin_severity</td>
<td><code>t_alert_level</code></td>
<td>TB_ERROR</td>
</tr>
<tr>
<td>setup_time</td>
<td>time</td>
<td>-1 ns</td>
</tr>
<tr>
<td>hold_time</td>
<td>time</td>
<td>-1 ns</td>
</tr>
<tr>
<td>bfm_sync</td>
<td><code>t_bfm_sync</code></td>
<td>SYNC_ON_CLOCK_ONLY</td>
</tr>
<tr>
<td>match_strictness</td>
<td><code>t_match_strictness</code></td>
<td>MATCH_EXACT</td>
</tr>
<tr>
<td>byte_endianness</td>
<td><code>t_byte_endianness</code></td>
<td>FIRST_BYTE_LEFT</td>
</tr>
<tr>
<td>valid_low_at_word_num</td>
<td>integer</td>
<td>0</td>
</tr>
<tr>
<td>valid_low_multiple_random_prob</td>
<td>real</td>
<td>0.5</td>
</tr>
<tr>
<td>valid_low_duration</td>
<td>integer</td>
<td>0</td>
</tr>
<tr>
<td>valid_low_max_random_duration</td>
<td>integer</td>
<td>5</td>
</tr>
<tr>
<td>check_packet_length</td>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>protocol_error_severity</td>
<td><code>t_alert_level</code></td>
<td>ERROR</td>
</tr>
<tr>
<td>ready_low_at_word_num</td>
<td>integer</td>
<td>0</td>
</tr>
<tr>
<td>ready_low_multiple_random_prob</td>
<td>real</td>
<td>0.5</td>
</tr>
<tr>
<td>ready_low_duration</td>
<td>integer</td>
<td>0</td>
</tr>
<tr>
<td>ready_low_max_random_duration</td>
<td>integer</td>
<td>5</td>
</tr>
<tr>
<td>ready_default_value</td>
<td>std_logic</td>
<td>'0'</td>
</tr>
<tr>
<td>id_for_bfm</td>
<td><code>t_msg_id</code></td>
<td>ID_BFM</td>
</tr>
</tbody>
</table>
Compiling UVVM Light

E.g. compiling from the /sim folder inside the UVVM Light install directory:

```
$ vsim -c -do "do ../script/compile.do [uvvm_light_directory] [target_directory]"
```

```
vsim -c -do "do ../script/compile.do ../ ."
```
BFM procedures are not sufficient

BFM: Defined here as a procedure only

- BFM procedures are great for simple testbenches
  - Dedicated procedures in a simple package
  - Just reference and call from a process
- BUT
  - A process can only do one thing at a time
    - Either execute that BFM
    - Or execute another BFM
    - Or do something else
- To do more than one thing:
  → Need an entity (or component)
  → (VC = Verification Component)

VVC: VHDL Verification Component (UVVM VC with extended functionality)

```
sbi_write(C_TX, x"B3")
uart_expect(x"B3")
```
VVC: VHDL Verification Component

**Interpreter**
- Is command for me?
- Is it to be queued?
- If not:
  - Case on what to do

**Executor**
- Fetch from queue
- Case on what to do
- Call relevant BFM(s) & Execute transaction

**Command Queue**

**Testcase Sequencer**

**SBI_VVC**

**UART (DUT)**
- Clocks
- Bus interface
- Other Ports
  - TX
  - RX
BFM to VVC: How?

UVVM VVCs also include:
Delay-insertion, command queuing, completion detection, activity registration, multicast & broadcast, termination, set-up, data fetch, multi-channel support, interface checkers, scoreboards, transaction info, local sequencers, etc ...

sbi_write(SBI_VVCT, 1, C_TX, x"B3")
uart_expect(UART_VVCT, 1, RX, x"B3")
sbi_write(C_TX, x"B3")
uart_expect(x"B3")
VVC based Testbench

```vlog
test_seq(sequencer);
axis_tx(target, data, ...);
axis_rx(target, data, ...);
```

VVC based Test harness

```
Clock-Gen VVC
AXI4-Stream Master VVC
DUT
AXIS slave
FIFO
AXIS master
AXI4-Stream Slave VVC
```

```
axistream_transmit(target, data, ...);
axistream_expect(target, data, ...);
```

BFM based Testbench

```
clock_generator
p_main(test-sequencer);
axis_tx(data, ...);
axis_rx(data, ...);
```

```
DUT
AXIS slave
FIFO
AXIS master
```
AXI-stream - VVC based TB (2)

- Full UVVM (all functionality)
- Dedicated library per VVC
  - For simpler reuse
- All VIP-related functionality in dedicated VIP directories
- Script to compile all UVVM
  - Compile all, but Just include what you need

UVVM (from github)

- `uvvm_util` (library)
  - `log`, `check_value`, `await_value`, etc...

- `bitvis_vip_clock_generator` (library)
  - `clock_generator_vvc` (VVC)
  - `start_clock`, ... (procedures / methods)
  - `clock_generator_vvct` (global signal)

- `bitvis_vip_axistream` (library)
  - `axistream_vvc` (VVC)
  - `axistream_transmit`, ... (procedures / methods)
  - `axistream_vvct` (global signal)
### Resulting transcript + Debug

#### Note the changing scope

```c
axistream_transmit(AXISTREAM_VVCT, 0, v_data_array, msg);
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Duration</th>
<th>Scope</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_UVVM_SEND_CMD</td>
<td>50.0 ns</td>
<td>TB seq.(uvvm)</td>
<td>axistream_transmit(AXISTREAM_VVC, 0, 512 bytes): 'TX 512B'</td>
<td>[6]</td>
</tr>
<tr>
<td>ID_PACKET_DATA</td>
<td>24202.0 ns</td>
<td>AXISTREAM_VVC, 0</td>
<td>axistream_transmit(512B) =&gt; Tx x&quot;ED&quot;, byte# 493. 'TX 512B'</td>
<td>[6]</td>
</tr>
<tr>
<td>ID_PACKET_COMPLETE</td>
<td>24346.0 ns</td>
<td>AXISTREAM_VVC, 0</td>
<td>axistream_transmit(512B) =&gt; Tx DONE. 'TX 512B'</td>
<td>[6]</td>
</tr>
</tbody>
</table>

```c
axistream_expect(AXISTREAM_VVCT, 1, v_exp_array, "Expecting **** ");
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Duration</th>
<th>Scope</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_UVVMSEND_CMD</td>
<td>50.0 ns</td>
<td>TB seq.(uvvm)</td>
<td>axistream_expect_bytes(AXISTREAM_VVC, 1, 512b): 'Expecting 512b'</td>
<td>[7]</td>
</tr>
</tbody>
</table>

- Plus similar additional verbosity as for Transmit
- Plus for both: Debug-messages when command reaches Interpreter and Executor
Documentation VVC

Similar docs for all BFM, VVCs, UVVM and other VIP
1 VVC procedure details

### Procedure

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>axstream_transmitt_bytes()</td>
<td>axstream_transmitt_bytes (VVC, vvc_instance_idx, data_array, [user_array, [strb_array, id_array, dest_array]], msg, [scope])</td>
</tr>
</tbody>
</table>

The `axstream_transmit()` VVC procedure adds a transmit command to the AXI4-Stream VVC when commands have completed. When the command is scheduled to run, the executor calls the `AXI4-Stream` BFM `QuickRef`. The `axstream_transmit()` procedure can only be called when the AXI STREAM VVC is instantiated with `GO_MASTER_MODE` to true.

### Examples:

- Syntax + Overloads
- Examples
- Explanations

3 VVC Configuration

#### Record element

<table>
<thead>
<tr>
<th>inter_bfm_delay</th>
<th>Type</th>
<th>C_AXISTREAM_BFM_CONFIG_DEFAULT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inter_bfm_delay</td>
<td>C_AXISTREAM_BFM_DELAY_DEFAULT</td>
<td>Delay between any requested BFM accesses towards the DUT. Add additional commands will result in an ERROR.</td>
</tr>
<tr>
<td>cmd_queue_count_max</td>
<td>natural</td>
<td>C_CMD_QUEUE_COUNT_MAX</td>
<td>Maximum pending number in command queue before queue is full. Adding additional commands will result in an ERROR.</td>
</tr>
<tr>
<td>cmd_queue_count_threshold</td>
<td>natural</td>
<td>C_CMD_QUEUE_COUNT_THRESHOLD</td>
<td>An alert with severity <code>cmd_queue_count_threshold</code> will be issued if command queue exceeds this count. Used for early warning if command queue is almost full. Will be ignored if set to 0.</td>
</tr>
<tr>
<td>result_queue_count_max</td>
<td>natural</td>
<td>C_RESULT_QUEUE_COUNT_MAX</td>
<td>Maximum number of untrusted results before result queue is full.</td>
</tr>
<tr>
<td>result_queue_count_threshold</td>
<td>natural</td>
<td>C_RESULT_QUEUE_COUNT_THRESHOLD</td>
<td>An alert with severity <code>result_queue_count_threshold</code> will be issued if result queue exceeds this count. Used for early warning if result queue is almost full. Will be ignored if set to 0.</td>
</tr>
<tr>
<td>result_queue_count_threshold_severity</td>
<td>t_alert_level</td>
<td>C_RESULT_QUEUE_COUNT_THRESHOLD</td>
<td>Severity of alert to be initiated if exceeding result_queue_count_threshold</td>
</tr>
<tr>
<td>bfm_config</td>
<td>t_axstream_bfm_config</td>
<td>C_AXISTREAM_BFM_CONFIG_DEFAULT</td>
<td>Configuration for AXI4-Stream BFM. See quick reference for AXI4-Stream BFM</td>
</tr>
</tbody>
</table>

The configuration record can be accessed from the Central Testbench Sequencer through the shared variable array, e.g.:

```solidity
shared_axstream_vvc_config(1).inter_bfm_delay.delay_in_time := 50 ns;
shared_axstream_vvc_config(1).bfm_config.clock_period := 10 ns;
```
Compiling UVVM

\script> vsim -c -do "compile_all.do"

The easiest way to compile the complete UVVM with everything (Utility Library, VVC Framework, BFMS, VVCs, etc.) is to go to the top-level script directory and run 'compile_all.do' inside Modelsim/Questasim/RivieraPro/ActiveHDL.
VVC: Easy to extend

- Easy to add local sequencers
- Easy to add checkers/monitors/etc

Interpreter
- Is command for me?
- Is it to be queued?
- If not: Case on what to do

Executor
- Fetch from queue
- Case on what to do
- Call relevant BFM(s) & Execute transaction

Command Queue

*_VVC

- Bit-rate checker
- Frame-rate checker
- Gap checker
VVC: Easy to extend

- Easy to handle split transactions
- Easy to handle out of order execution

**Interpreter**
- Is command for me?
- Is it to be queued?
- If not: Case on what to do

*VVC*

**Command Queue**

**Executor**
- Fetch from queue
- Case on what to do
- Call relevant BFM(s) & Execute transaction

**Bit-rate checker**

**Frame-rate checker**

**Gap checker**

**Queue**

**Response-Executor**
VVC Advantages

- Simultaneous activity on multiple interfaces
- Encapsulated \(\rightarrow\) Reuse at all levels
- Queue \(\rightarrow\) May initiate multiple high level commands
- Local Sequencers for predefined higher level commands

**Only in UVVM VVCs:**
- UNIQUE: Control all VVCs from a single sequencer!
- May insert delay between commands – from sequencer
  \(\rightarrow\) The only system to target cycle related corner cases
- Simple handling of split transactions and out of order protocols
- Common commands to control VVC behaviour
- Simple synchronization of interface actions – from sequencer
- May use Broadcast and Multicast

Better Overview, Maintenance, Extensibility and Reuse
Keeping the overview

- May use any number of VVCs
- May use any number of instances of each VVC type
- May control them all simultaneously – and also control command delays
- May control all from a single test sequencer (or two – or more)
- Get total overview by looking at one file of sequential commands only
Lot’s of free UVVM BFM$s and VVC$s

- AXI4-lite
- AXI4 Full
- AXI-Stream Transmit and Receive
- UART Transmit and Receive
- SBI
- SPI Transmit and Receive
- I2C Transmit and Receive
- GPIO
- Avalon MM
- Avalon Stream Transmit and Receive
- RGMII Transmit and Receive
- GMII Transmit and Receive
- Ethernet Transmit and Receive
- Wishbone
- Clock Generator
- Error Injector

All:
- Free
- Open Source
- Well documented
- Example Testbenches

The largest collection of VHDL Interface Models

VVC: VHDL Verif. Comp.
- Includes the corresponding BFM
- Simultaneous interface handling
- Synchronization of interfaces
- Skewing between interfaces
- Additional protocol checkers
- Local sequencers
- Activity detection
- Simple reuse between projects
The newer stuff – in cooperation with ESA

- ESA Extensions in ESA-UVVM-1
  - **Scoreboards**
  - Monitors
  - Controlling randomisation and functional coverage
  - Error injection (Brute force and Protocol aware)
  - Local sequencers
  - Controlling property checkers
  - **Watchdog** (Simple and Activity based)
  - **Transaction info**
  - Hierarchical VVCs - And Scoreboards for these
  - **Specification Coverage** (Requirement/test coverage)

ESA is helping VHDL designers speed up FPGA and ASIC development and improve their product quality!
Transaction info transfer

Transaction info  Inside VVC  All UVVM VVCs
Generic Scoreboard

- Statistics
- Compare
- Queue

Expected data ➔ Statistics ➔ Actual data

- Configuration record:
  - allow_lossy
  - allow_out_of_order
  - mismatch_alert_level
  - etc...

- Counting:
  - entered
  - pending
  - matched
  - mismatched
  - dropped
  - deleted
  - initial garbage

- generic data type
  - logging/reporting
  - flushing queue
  - clearing statistics

- insert, delete, fetch
- ignore_initial_mismatch
- indexed on either entry or position
- optional source element (in addition to expected + actual)

Quick Reference is provided

UVVM - Bringing UVM to VHDL

EmLogic
Advanced scoreboard-based TB

VVC based Testbench

- \texttt{p\_main (test-sequencer)}
- \texttt{... axis\_tx(target, data, ...); axis\_rx(target, data, ...); ...}

VVC based Test harness

- AXI4-Stream Master VVC
- Clock-Gen VVC
- DUT
- 
- AXI4-Stream Slave VVC
- AXI4-Stream Scoreboard

UVVM - Bringing UVM to VHDL

\begin{verbatim}
axistream_transmit(AXISTREAM_VVCT,0, v_data_array, msg);
\end{verbatim}

\begin{verbatim}
axistream_receive(AXISTREAM_VVCT,1, v_data_array, "Checking via SB");
\end{verbatim}
Watchdogs

Simple WD  Inside Util

Watchdog

watchdog_timer(watchdog_ctrl, timeout, [alert_level, [msg]])
extend_watchdog(watchdog_ctrl, [time_extend])
reinitialize_watchdog(watchdog_ctrl, timeout)
terminate_watchdog(watchdog_ctrl)

Activity WD  VVCs and UVVM

activity_watchdog(timeout, num_exp_vvc);

Apply both concurrently
Assure that all requirements have been verified

1. Specify all requirements

<table>
<thead>
<tr>
<th>Requirement Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOTOR_R1</td>
<td>The acceleration shall be ***</td>
</tr>
<tr>
<td>MOTOR_R2</td>
<td>The top speed shall be given by ***</td>
</tr>
<tr>
<td>MOTOR_R3</td>
<td>The deceleration shall be ***</td>
</tr>
<tr>
<td>MOTOR_R4</td>
<td>The final position shall be ***</td>
</tr>
</tbody>
</table>

2. Report coverage from test sequencer (or other TB parts)
3. Generate summary report

- Solutions exist to report that a testcase finished successfully
  - BUT - reporting that a testcase has finished is not sufficient
What if multiple requirements are covered by the same testcase?

- E.g. Moving/turning something to a to a given position
  - R1: Acceleration
  - R2: Speed
  - R3: Deceleration
  - 4: Position etc..

Generates various types of reports
- Coverage per requirement
- Test cases covering each requirement
- Requirements covered by each Test case
- Accumulated over multiple Test cases
The brand new stuff – October 2021

- Enhanced Randomisation
  - Advanced randomisation in a simple way

- Optimised Randomisation
  - Randomisation without replacement
  - Weighted according to target distribution AND previous events
  → the lowest number of randomisations for a given target

- Functional Coverage
  - Based on functional coverage in SV
    - But in VHDL, and without all the complexity of SV and UVM
  - Fully integrated with UVVM, but may be used stand-alone
Well integrated with UVVM

- Alert handling and logging in particular
- Strong focus on Overview & Readability

- Adding keywords to ease understanding
- Easy to Maintain and Extend

Typing code consumes is an insignificant part of the development time.

Reading and understanding code is repeated over and over again, and is thus a significant part of the development time

```
addr <= my_addr.rand(0, 18, ADD,(30,31), EXCL(7));
```

➔ Investing in better code yields a huge return on investment
Single Method approach

- "Standard" approach: Randomisation in one single command
  - Simple randomisation is always easy to understand
    
    ```vhdl
    addr <= my_addr.rand(0, 18);
    ```
  - More complex randomisation is normally more difficult to understand
    BUT – there are ways to significantly improve this
    
    ```vhdl
    addr <= my_addr.rand(0, 18, EXCL,(7));
    addr <= my_addr.rand(0, 18, ADD,(30,31));
    addr <= my_addr.rand(0, 18, ADD,(30,31), EXCL,(7));
    ```
  - Similar readability focus for weighting
    
    ```vhdl
    addr <= my_addr.rand_val_weight((0,2),(1,3),(2,5));
    addr <= my_addr.rand_range_weight((0,18,4),(19,31,1));
    ```
Multi-method approach (1)

- Extends the functionality of the single method approach
  - Single method approach:
    ```vhdl
    addr_1 <= my_addr.rand(0, 18, ADD,(30,31), EXCL,(7));
    addr_2 <= my_addr.rand(0, 18, ADD,(30,31), EXCL,(7));
    ```
  - Multi-method - equivalent
    ```vhdl
    my_addr.add_range(0, 18);
    my_addr.add_val((30,31));
    my_addr.excl_val((7));
    addr_1 <= my_addr.randm(VOID);
    addr_2 <= my_addr.randm(VOID);
    ```
    Note: `randm()` (For clarity and to avoid any ambiguity)
  - Allows adding more ranges, sets or exclusions
    ```vhdl
    my_addr.add_range(48,63);
    my_addr.add_range(80,127);
    ```
  - Allows simple inclusion of future extensions
Functional Coverage – Typical Sequence

- Define a variable of type `t_coverpoint`
  ```
  variable cp_payload_size : t_coverpoint;
  ```

- Add the bins
  ```
  cp_payload_size.add_bins(bin(0));
  cp_payload_size.add_bins(bin(1));
  cp_payload_size.add_bins(bin_range(2,254,1));
  cp_payload_size.add_bins(bin(255,256,2));
  ```

- Tick off bins as their corresponding payload size is used
  ```
  cp_payload_size.sample_coverage(payload_size);  
  ```

- Continue sending packets until coverage target is reached
  ```
  while not cp_payload_size.coverage_completed(VOID);
  ```

UVVM also has transition coverage
Some reports – out of many

### Coverage Summary Report (NON VERBOSE)

```
Coverpoint:  Covpt_1
Coverage (for goal 100): Bins: 60.00%, Hits: 76.47%
```

<table>
<thead>
<tr>
<th>BINS</th>
<th>HITS</th>
<th>MIN HITS</th>
<th>HIT COVERAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(256 to 511)</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>(0 to 125)</td>
<td>6</td>
<td>8</td>
<td>75.00%</td>
</tr>
<tr>
<td>(126, 127, 128)</td>
<td>3</td>
<td>1</td>
<td>100.00%</td>
</tr>
<tr>
<td>(129 to 255)</td>
<td>14</td>
<td>4</td>
<td>100.00%</td>
</tr>
<tr>
<td>(0-&gt;1-&gt;2-&gt;3)</td>
<td>0</td>
<td>2</td>
<td>0.00%</td>
</tr>
<tr>
<td>transition_2</td>
<td>2</td>
<td>2</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

### Overall Coverage Report (VERBOSE)

```
OVERALL COVERAGE REPORT
```

| COVERPOINT | COVERAGE | WEIGHT | COVERED BINS | COVERAGE(BINS|HITS) | GOAL(BINS|HITS) | % OF GOAL(BINS|HITS) |
|------------|----------|--------|--------------|----------------|--------------|----------------|
| Covpt_1    | 1        | 3 / 5  | 60.00% | 76.47%        | 50% | 100% | 100.00% | 76.47%       |
| Covpt_2    | 1        | 3 / 3  | 100.00% | 100.00%       | 100% | 100% | 100.00% | 100.00%      |
| Covpt_3    | 1        | 6 / 6  | 100.00% | 100.00%       | 100% | 100% | 100.00% | 100.00%      |
| Covpt_4    | 1        | 0 / 4  | 0.00%  | 0.00%         | 100% | 100% | 100.00% | 0.00%        |
| Covpt_5    | 1        | 0 / 1  | 0.00%  | 0.00%         | 100% | 100% | 100.00% | 0.00%        |
| Covpt_6    | 1        | 4 / 4  | 100.00% | 100.00%       | 100% | 100% | 100.00% | 100.00%      |
| Covpt_7    | 1        | 0 / 3  | 0.00%  | 0.00%         | 100% | 100% | 100.00% | 0.00%        |
| Covpt_8    | 1        | 12 / 12| 100.00% | 100.00%       | 100% | 100% | 100.00% | 100.00%      |
Pick and choose – No lock

- Pick **any** Utility Library functionality: (from these plus more)
  
  | log() | alert() | error() | manual_check() |
  | check_value() | check_stable() | await_stable() |
  | await_change() | await_value() | check_value_in_range() |
  | random() | randomize() | report_***() | enable_log_msg() |
  | justify() | fill_string() | to_upper() | replace() |
  | clock_generator() | await_unblock_flag() | await_barrier() |

- Pick any BFM - or any VVC – or any combination
  
  AXI4-lite  GPIO  SBI  SPI  UART  I2C
  AXI  AVALON MM  AXI4-stream  Avalon-stream
  CLOCK_GENERATOR  GMII  RGMII  Ethernet

- Pick any FIFO, Queue, Scoreboard
- Pick any Advanced Randomisation and/or Functional Coverage
- Pick Specification coverage / Requirements tracking
Standardized? – In what way?

- Standard Interface
- Standard Protocol
- Standard common commands
- Standard Status interface
- Standard Config interface
- Standard handling of multiple VVCs
- Standard VVC synchronization
- Standard multicast/broadcast

Simplification

VVCs from different users will work together

Users know how VVCs behave and how any test harness will work
UVVM vs UVM

- UVVM: Component oriented vs UVM: Object oriented
- Block diagrams are similar, but different naming and structure
- UVM is far more comprehensive and complex than UVVM
  - But UVVM is sufficient for almost all testbenches
- UVVM user threshold is a fraction of the UVM threshold – for VHDL users
  - UVVM is just a step-by-step evolution on VHDL
- UVVM allows a gentle introduction to modern verification
  - May be used as a first step to UVM – for those who evaluate that
  - Is however sufficient in itself for almost all FPGA designs
- UVVM can run on any VHDL 2008 compatible simulator
  - Is included with Modelsim, Questa and Riviera-PRO
UVVM in a nutshell

- Huge improvement potential for more structured FPGA verification

UVVM (incl. all) is Open Source

Game changer for efficiency & quality

UVVM has the largest collection of interface models (as BFM and VVCs)

UVVM may save 200-2000 hours on a medium complex project
And at the same time improve TTM, MTBF & LCC

Usage is exploding
- World-wide number 1 for VHDL
- Fastest growing – of all
Courses

- FPGA (and ASIC) Verification: 'Advanced VHDL Verification – Made simple'
- FPGA (and ASIC) Design: 'Accellerating FPGA and Digital ASIC Design'
- Courses on demand/request anywhere: On-site, Online, Public,

**Design**
- Design Architecture & Structure
- Clock Domain Crossing
- Coding and General Digital Design
- Reuse and Design for Reuse
- Timing Closure
- Quality Assurance - at the right level
- Faster and safer design

**Verification**
- Verification Architecture & Structure
- Self checking testbenches
- BFMs – How to use and make
- Checking values, time aspects, etc
- Verification components
- Advanced Verif: Scoreboard, Models, etc
- State-of-the-art verification methodology

https://emlogic.no/courses/