# OSVVM SUMMARY

* OSVVM Verification Framework:

A diagram of a computer

AI-generated content may be incorrect.

A diagram of a diagram

AI-generated content may be incorrect.

* **Verification Components (VCs)** for interfacing with the DUT (AXI Manager, UART, Memory models etc.).
* A **Test Sequencer (TestCtrl)** that drives the test behavior via transaction calls. Each test case is a separate architecture of TestCtrl.
* A **Transaction Interface (record)** and **Transaction API (procedures)** to communicate between the sequencer and VCs.

A screenshot of a computer code

AI-generated content may be incorrect.

* This is an example of a transaction interface (record). The record is an inout port.

A screenshot of a computer program

AI-generated content may be incorrect.

* This is an example of a Transaction API (a procedure).
* OSVVM defines common Transaction APIs for interfaces:
* **AddressBus** (e.g., AXI, Avalon): Write, Read
* **Streaming** (e.g., AXI Stream, UART): Send, Get
* For these interfaces, **Model Independent Transactions (MIT)** defines Transaction Interface (record) and Transaction API (procedures).
* MIT speeds up VC and test creation, and enables reuse.

A screenshot of a computer program

AI-generated content may be incorrect.

* This is an example of a Verification Component. The VC waits for a transaction, decodes the operation (READ, WRITE) and executes appropriate behaviour that corresponds to an axi\_read or axi\_write operation.

A yellow box with black text

AI-generated content may be incorrect.

* This is an example of a Test Controller. The Test Controller includes ports for all transaction records and one architecture per test case.
* **Constrained Random Testing:** enables generating varying test scenarios using random integers (with constraints or exclusions) or weighted distributions.

A diagram of a software system

AI-generated content may be incorrect.

* **Scoreboards:** Internally, it is a FIFO and a checker.
* The expected result is pushed to the scoreboard by generating a transaction like Send(…). The data is received using Get(…) and it is checked against what was previously pushed into the scoreboard via Check(…).
* **Functional Coverage:** helps tracking what features and scenarios has been tested up to now. It is a different aspect of testing than code coverage.
* Item Coverage (Point Coverage): Tracks values of a single variable or feature.
* Cross Coverage: Tracks combinations of values across multiple variables.
* CoveragePkg
* Define bins using GenBin() and call Icover(ID, value) to mark a bin as covered.
* **OSVVM Libraries:**
* Utiliy Library: Logging, alerts, coverage, randomization
* AXI4, UART and RAM Libraries: Ready-to-use verification components
* Library osvvm; context osvvm.OsvvmContext;
* **Scripting API:** OSVVM includes a tool-independent scripting API
* Works with various simulators
* Uses .pro file extensions
* Facilitates library setup, compilation, simulation
* **Test Reports:** OSVVM includes automated, readable and structured reports to track test and verification progress.
* Build Summary Report in HTML: High-level view of the whole test
* Test Case Reports in HTML: Details in verification components, alerts, coverage, scoreboard and logs
* Junit XML Report: For Continuous Integration Tools

# UVVM SUMMARY

* UVVM (Universal VHDL Verification Methodology)
* Free, open-source and structured VHDL verification library and methodology
* Compliant with VHDL-2008 simulators
* Developed by EmLogic, which is a Scandinavian FPGA design center.
* Reusable and maintainable components
* **Utility Library:** ready-to-use procedures and functions for clear and readable testbenches. Simplifications for signal checking, clock generation and more.
* check\_value(signal, expected, message)
* check\_stable(signal, duration, message)
* gen\_pulse(signal, value, duration, message)
* await\_value(signal, expected, offset, timeout, message)
* clock\_generator()
* Randomization: random(), randomize()
* **Bus Functional Models (BFMs):** simulates interfaces like UART, SPI, I2C, AXI etc. in testbenches by managing the corresponding transactions. They are useful to communicate with the DUT through commonly used protocols.
* They simulate a single transaction (read, write, expect, etc.).
* Cannot execute more than one transaction at a time.
* UART, AXI4-FULL, AXI4-LITE, AXI-STREAM, SPI, I2C, GPIO, SBI, Avalon etc.
* uart\_transmit(x"2A"); -- Send value via UART RX as an example
* **VHDL Verification Components (VCCs):** An upgrade of BFMs. BFMs are used via procedure calls so they are sequential and blocking. VCCs can do more than one thing at a time. An entity or component is needed for a VCC.
* Handle concurrent activity on multiple interfaces
* Each VVC includes: Interpreter, Command Queue, Executor
* **Scoreboards:** Used for comparing actual output from DUT against expected values. The expected values are generated by the testbench code or from a reference model.
* The scoreboards match entries ,detect and report mismatches.
* Works with generic data types.
* Are able to track initial garbage.
* Final scoreboard report shows counts of passed, failed, pending, etc.
* There are advaned features such as accepting some loss in data by allow\_lossy which is used in streaming projects.
* **Functional Coverage:** Used for verifying if the testbench has stimulated all the necessary inputs and scenarios. Helpful for observing the operation for border cases.
* Define specific input scenarios by assigning them to different bins.
* Track which bins have been triggered during simulation.