

# Universal Verification Methodology (UVM)-based SystemVerilog Testbench for VITAL Models

by Tanja Cotra, Program Manager, HDL Design House

With the increasing number of different VITAL model families, there is a need to develop a base Verification Environment (VE) which can be reused with each new VITAL model family.

UVM methodology applied to the SystemVerilog Testbench for the VITAL models should provide a unique VE. The reusability of such UVM VE is the key benefit compared to the standard approach (direct testing) of VITAL models verification. Also, it incorporates the rule of “4 Cs” (Configuration, Constraints, Checkers and Coverage). Thus, instead of writing specific tests for each DUT feature, a single test can be randomized and run as part of regression which speeds up the collection of functional coverage.

The results show that UVM VE testbench, with respect to a standard direct test bench, requires nearly equal time to develop. In return it provides re-usability and much faster verification of each new VITAL model. The changes one needs to do are mainly related to the test where the appropriate configuration must be applied.

The prevailing method in verification of VITAL models was based on the use of a direct testbench where we can point out two basic problems:

- 1) For each model, a new testbench needs to be developed, which is time consuming, and may only be used with that specific VITAL model.
- 2) The direct testing does not provide functional coverage information as the main parameter for the overall verification progress.

In order to overcome the issues above, the answer was to migrate to UVM which is based on the well proven OVM Verification Methodology. The UVM methodology applied to the SystemVerilog Testbench for VITAL models should provide a unique VE that can be reused later with minimal changes.

The initial version of the SystemVerilog VITAL testbench, which is based on UVM, is intended for verification of serial

flash family of VITAL models. One serial flash VITAL model contains a set of specific instructions which is common for all models belonging to the serial model family.

Having reusable verification components, we can significantly reduce the time needed to set the environment for verification of each new serial flash model. By incorporation of “Three Cs” rule (Constraints, Checkers and Coverage), instead of writing specific tests for each DUT feature, a single test can be randomized and run as part of a regression which speeds up the collection of functional coverage.

The main parts of our UVM environment are:

- Top module
- Test
- Configuration class
- Testbench class
- Environment class

The top module instantiates device-under-test (DUT) and DUT interface, used for connecting the VE with the DUT. Also, the top module generates the necessary clock signal and calls the predefined UVM task `run_test()`, which is used for running the specific test. The name of the specific test must be provided on the command line.

Since all verification components are defined as classes, they cannot be directly connected to the actual DUT interface but rather through the construct of virtual interface. The virtual interface is a SystemVerilog type and it is instantiated inside a specific VE component which has the need to access some signals on the actual interface (such components are the driver and monitor, for example). The actual interface is made visible to all components through the use of a predefined configuration table.

The main purpose of the configuration table is to parametrize the VE components so they can be easily customized from the specific test. Since UVM does not allow the interface to be directly added to the configuration table, a wrapper is defined around each interface.

This wrapper is stored in the configuration table by using the `set_config_object()` method. This enables every component, which needs access to the interface, to be able to retrieve it from the configuration table by using the `get_config_object()` method.

The wrapper is a user defined class which extends `uvm_object`. It contains the instance of the virtual interface and its constructor takes the virtual interface as an argument. At the top level, where the actual interface is instantiated, this wrapper is also instantiated where, its constructor is called and the interface is passed in as an argument. This wrapper is added to the configuration table. Code example would be:

```
class dut_if_wrapper extends uvm_object;
    virtual dut_interface dut_if_vi;
    function new (string name, virtual dut_interface arg)
        super.new();
        dut_if_i = arg;
    endfunction: new
endclass

module top;
    ...
    dut_interface dut_if ();

    initial begin
        dut_if_wrapper if_wrapper =
            new("if_wrapper",dut_if);
        set_config_object("**","dut_if_wrapper",
            if_wrapper,0);
        ...
        run_test();
    end
endmodule
```

The `set_config_object()` sets the object into the configuration table. The first argument of the `set_config_object()` method is `*`, which is the hierarchical path of the VE component for which we are setting information. In this case, the wildcard `*` makes the DUT interface available to the entire VE. The second argument is the name of

the configuration parameter that we are setting. The third argument is the value of that parameter. The last argument determines if we are adding only the reference to the object (0) or to the actual object (1). The configuration parameter stored can then be retrieved inside the `build()` method of any component by using the `get_config_object()`. In this way, every component gets access to the actual DUT interface. This verification methodology simplifies accessing DUT's signals and significantly improves the verification environment's reusability.

Each test contains an instance of the configuration class and testbench class. Inside the test, the object of the configuration class is constructed, and if necessary randomized. Then it is added to the configuration table.

The configuration, as "4th C", is implemented through a configuration class. The fields of this class represent the values which will change when a new serial flash VITAL model appears. In this case, the fields are timing parameters (setup and hold times, for example) which will change with each new model. Similar to the interface wrapper, `set_config_object()` and `get_config_object()` are also used for this class, so that any component in the environment can retrieve it, if needed. It is set during the build phase of the test, so the new DUT will only be required to set different configuration parameters for the test without changing the rest of the environment.

The testbench class, consists of the following blocks:

- 1) Scoreboard
- 2) Coverage collector
- 3) Environment

Inside the `build_phase()` method, these three components are created. Also, the testbench extends two predefined methods:

```
connect_phase() -
    to subscribe the scoreboard and coverage collector
    to monitor analysis port
end_of_elaboration_phase() -
    to set report verbosity level and to print
    testbench topology.
```

To provide checkers and coverage, two components are subscribed to the monitor: scoreboard (which performs data checks) and coverage\_collector (which contains SystemVerilog coverage groups).

The scoreboard class is defined by extending the base class `uvm_scoreboard`. The scoreboard represents a type of data checker as it checks for the written data integrity, erasure of particular locations, erasure of whole memory, read operation results, register access, etc.

The scoreboard collects information on the DUT's inputs. Depending on the data driven into the DUT, the DUT's functional specification and the current configuration, the expected output data is calculated and placed in the scoreboard list. When the output data from the DUT is collected, the data checker part of the scoreboard checks whether the DUT's output data matches the expected scoreboard's data. The scoreboard is a crucial block since it models the DUT behavior.

The scoreboard contains the memory model, resembling its organization (banks, sectors, subsectors). This memory model is preloaded with the same data as the DUT.

The scoreboard receives the monitor's transactions. When data is being written into a memory location, the monitor sends this data to the scoreboard, which in turn provides that this data is written to the memory model.

In the case of reading data, the data checker compares the DUT's data with the memory model's scoreboard data.

During the erase operation, the erase checker checks if both the memory model and the DUT's memory were erased.

This testbench environment allows for functional coverage to be collected by the use of a coverage collector component. The coverage collector uses the collected bus transactions from the monitor and checks if all of the DUT's features are covered. Coverage groups include different coverage items, checking if all of the cover items relevant combinations were covered by the transactions.

For example, coverage group `read_cg` checks if all locations have been read; coverage group `write_cg` checks if all locations have been programmed.

All of the necessary coverage groups are defined within the verification environment in order to check the device's functionality.

The environment class instantiates only the agent component. Since data needs to be driven to the DUT, this environment requires an active agent which instantiates the sequencer, driver and monitor.

The DUT being verified with this VE is a memory with Serial Peripheral Interface (SPI). The main transaction class, by which all sequences are parametrized, is defined so that it contains all the necessary information for memory access (`instruction_type`, `address`, `data`, etc.). It has constraints to keep the values inside allowed ranges as specified by the DUT protocol. For example, the number of address bytes sent to the DUT depends on the instruction. If READ is issued, constraints keep the number of address bytes according to protocol to either 3 or 4.

The constraints are an important part of the environment. They are specified inside the main transaction, and also inside sequences and tests. The idea is to set constraints in the test on a specific sequence, and then all lower constraints (inside sequences and the main transaction) are set automatically.

All sequences are written and stored inside a separate file called `seq_lib.sv`. Inside the specific test one or more sequences from the `seq_lib` file are created, randomized if necessary, and started on the sequencer. For example, the read sequence contains the fields: `address`, `number_of_address_bytes` and `number_of_read_bytes`. These fields are prefixed with `rand`, but also they are kept in reasonable range by using constraints. This approach enables the creation of a smaller number of sequences with random fields instead of manually writing a large number of specific sequences in a standard testbench.

When a new serial flash VITAL model appears, the changes can be made from the test by setting the appropriate configuration. For example, a new serial flash model will have different timing parameters. From the test, the inline constraint is used to set appropriate parameters:

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    spi_configuration =
spi_configuration_c::type_id::create("spi_configuration");
    ...
    assert(spi_configuration.randomize() with { ... });
    set_config_object("**", "spi_configuration",
spi_configuration, 0);
    spi_configuration.print();
endfunction: build_phase
```

Basically, all parameters that change with the new serial flash model, can be added to the configuration class and set appropriately from the test. Although this VE cannot be completely reusable with other families of VITAL models, it still offers a certain amount of reusability through UVM features like instance and type overrides.

The results show that this kind of testbench initially requires more time to develop, but in return provides re-usability and much faster verification of each new serial flash VITAL model.