Simulation environment based on the Universal Verification Methodology

A. Fiergolski on behalf of the CLICdp collaboration

The European Organization for Nuclear Research, Geneva, Switzerland

E-mail: Adrian.Fiergolski@cern.ch

ABSTRACT: Universal Verification Methodology (UVM) is a standardized approach of verifying integrated circuit designs, targeting a Coverage-Driven Verification (CDV). It combines automatic test generation, self-checking testbenches, and coverage metrics to indicate progress in the design verification. The flow of the CDV differs from the traditional directed-testing approach. With the CDV, a testbench developer, by setting the verification goals, starts with a structured plan. Those goals are targeted further by a developed testbench, which generates legal stimuli and sends them to a device under test (DUT). The progress is measured by coverage monitors added to the simulation environment. In this way, the non-exercised functionality can be identified. Moreover, the additional scoreboard indicates undesired DUT behavior. Such verification environments were developed for three recent ASIC and FPGA projects which have successfully implemented the new work-flow: (1) the CLICpix2 65 nm CMOS hybrid pixel readout ASIC design; (2) the C3PD 180 nm HV-CMOS active sensor ASIC design; (3) the FPGA-based DAQ system of the CLICpix chip.

This paper, based on the experience from the above projects, introduces briefly UVM and presents a set of tips and advices applicable at different stages of the verification process-cycle.

KEYWORDS: Digital electronic circuits; VLSI circuits
1 UVM concept

Universal Verification Methodology (UVM) is a standardized method for verifying integrated circuit designs [1]. Its first version was introduced in 2011. Unlike the previous methodologies developed independently by the simulator vendors, it is an Accellera [2] standard supported by multiple vendors.

The concept behind the UVM targets a Coverage Driven Verification (CDV). It combines automatic test generation, self-checking testbenches, and coverage metrics to indicate progress in the design verification. The flow of the CDV differs from the traditional directed-testing approach. With the CDV, a testbench developer, by setting the verification goals, starts with an organized plan. Those goals are targeted further by a developed testbench, which generates legal stimuli and sends it to a Device Under Test (DUT). The progress is measured by coverage monitors added to the simulation environment. In this manner, the non-exercised functionality can be identified. Moreover, the additional scoreboards indicate undesired DUT behaviour.

A thorough design validation can be obtained by a change of testbench parameters and randomization seeds. To meet the verification goals quicker, the simulation can be tuned by test constraints added on top of the testing environment. While the CDV supports directed and constrained-random approaches, the combination of both is preferred: a user can let the constrained-random testing do most of the work before devoting to writing the time-consuming, deterministic tests aiming at a specific scenario, which is difficult to reach randomly.
1.1 Testbench architecture

A UVM testbench is composed of verification components — encapsulated, reusable, ready-to-use, configurable elements checking an interface protocol, a design sub-module, or a full system. The architecture of each component is logical. It includes a complete set of elements enabling the stimulation, check and collection of coverage information related to the specific protocol or design.

![Figure 1. Typical UVM testbench architecture [1].](image)

An example of a test environment is presented in figure 1. A developer reuses three interface verification components from a common set, instantiating and configuring them in a required operational mode. The timing and data correlation between the different interfaces, as well as control of the test environment in the particular testbench, is obtained through a multi-channel sequence mechanism — a virtual sequencer.

1.2 Transactions

Transactions represent an input and output to/from the DUT, e.g. network packets, bus accesses, instructions, etc.. Their fields and attributes are derived from a specification of a given interface, e.g. the Ethernet standard defines the valid values and attributes for the Ethernet packets. Unlike the tests based on a fully random generation of transactions and the DUT stimulation, the UVM suggests a targeted randomization of transaction fields by use of System Verilog constraints. This approach yields a larger number of meaningful stimuli maximizing the overall coverage.

1.3 Driver

The actual active emulation of a logic stimulating DUT is handled by a driver. It repeatedly receives transactions and executes them by sampling and driving the DUT ports. This implies that to perform a read transfer on a generic bus the driver will control all involved data lines, address lines and read/write strobes.
1.4 Sequencer

The transactions are produced, controlled and provided to the driver by a sequencer. Its basic behaviour, the generation of a randomly filled transaction on request of the driver can be extended. By applying the System Verilog constraints directly on the transaction, the sequencer has the possibility to control the distribution of the randomized values. The UVM extends the sequencer’s capabilities further by implementing a set of features available to the developer out-of-the-box. The UVM sequencer has the ability to react on the current state of the DUT for every generated transaction. Moreover, by defining a sequence with a strict order between the generated transactions, the user has the possibility to obtain a structured, meaningful and longer stimulus pattern. Such user-defined sequences can be reused in different test scenarios. Additionally, the use of a virtual sequencer, as shown in figure 1, allows for synchronization at the design level and control of multiple verification components. The layered architecture of the sequencers facilitates modelling of a complex multi-layered protocol.

1.5 Monitor

Even though the drivers and sequencers control traffic on the particular interface, they are not used for coverage and checking. The task of sampling DUT signals is delegated to a passive module — the monitor. It extracts the interface traffic and translates it into transactions, that may be available to other verification components, such as scoreboards. The same applies to the traffic generated by a driver itself — it is captured by the monitor. The monitor allows to spy at a state of the DUT and broadcast it to the test environment. It enables event-driven simulation: the virtual sequencer spawns a sequence on a certain interface after notification from the monitor about occurrence of a defined event on another bus.

The consistency checks of the traffic on the DUT lines and its validation against the protocol specification lies in the domain of the monitor. Moreover, it collects coverage of the given interface. Optionally, the monitor may print trace information.

1.6 Agent

Although the sequencers, drivers and monitors can be reused independently, in order to improve the interoperability, the UVM methodology recommends encapsulation of a related driver, sequence and monitor in a more abstract container called an agent. A single agent can stimulate and verify the DUT. However, the verification components, similarly to the ones shown in figure 1, may contain a few agents. The agent can initiate the transactions to the DUT (master) or react to the transaction requests (slave). The UVM distinguishes either active or passive agents. The active agent stimulates the DUT by driving transactions according to the test scenario and monitors the device. The passive agent, having enabled only the monitor, is limited to the DUT sampling activity.

1.7 Scoreboard

The scoreboard performs higher-level validation checks based on the transactions received from the monitors of the different verification components. It may contain a functional model of the DUT. In this way, collecting traffic from all DUT ports, the scoreboard enables verification of the device behaviour.
1.8 Environment

A higher-level verification component is called an environment. An example is shown in figure 2. It may consist of many agents, as well as other components, like monitors or virtual sequencers. The configuration of the environment enables customization of its topology and behaviour according to the specific UVM test. The function of a single environment is the generation of the constrained-random traffic to stimulate the DUT, monitoring of the DUT response, check of ongoing traffic with respect to protocol specifications and the coverage collection.

![Figure 2. Example of a verification environment.](image)

1.9 Test

The UVM test is a top level element of the UVM based simulation. It defines a testbench scenario, by scheduling execution of the high level sequences on the respective virtual sequencers. Moreover, the UVM test enables configuration of its verification components and customization of the reusable environments.

1.10 The Class Libraries

The UVM Class Library provides System Verilog base classes, utilities and macros which facilitate development of the well-constructed, reusable testbench following the CDV. The introduced verification components enable encapsulation and hierarchical instantiation. The communication between them is based on the Transaction Level Modelling (TLM) [1].

The control of the verification components is achieved through an extendible set of phases to initialize, run and complete each UVM test. Although the base phases are defined in the library, they can be extended to meet specific design requirements.
The efficiency in UVM usage originates from the verification libraries provided by the Electronics Design Automation (EDA) vendors. A user obtains agents of many common interfaces like Ethernet, PCIe, USB, \textit{I^2}C, etc. with a set of assertions and checkers. The potential misbehaviour of the DUT is automatically spotted and a warning indicating a violated paragraph of the given interface specification is printed.

More information about the UVM library and the methodology itself can be found in references [1] and [3].

2 Verification

A systematic verification based on UVM takes place in three stages: specification of a test plan (section 2.1), development of a verification platform (section 2.2) and actual verification (section 2.3).

Each step can be facilitated by an appropriate EDA tool. The hints provided in sections 2.1–2.3, below, directly refer to Questa workflow from Mentor Graphics [6], which is available for academic purposes through the EUROPRACTECE platform [8]. However, the hints given also apply to the solutions provided by other EDA vendors.

2.1 Verification plan

At the stage of the verification plan preparation, the verification engineer and the DUT developers evaluate the DUT specifications in order to identify properties of the device which should be addressed by tests. The extracted verification points should be documented, as a text document or a spreadsheet, for future reference.

![Figure 3](image)

An example verification plan is presented in figure 3. The columns shown refer to the sections of the DUT specification, title of the verification point, its description, link, type, the weight and the target coverage goal for a given point. At the first iteration, when the focus is on the extraction of all verification points, the implementation of observers representing the verification goals in the source code is not yet performed. Thus, the link and type columns should remain empty at first. Later, the verification engineer fills the type column with the appropriate implementation of the coverage model for each point of the plan. System Verilog provides a rich set-of functional-coverage

---

---
features. In terms of explicit (user-defined) coverage, the developer chooses between cover group and cover property modelling. The first one checks permutations of condition and state when a known result is achieved (System Verilog Covergroups and Coverpoints). The second one indicates that a set of state transitions has been observed (System Verilog Assertions). While implementing the verification platform (section 2.2), the links to the instances of the coverage observers in the verification environment are added to the test plan.

The EDA vendors provide tools which facilitate the specification of the test plan. Questa from Mentor Graphics comes with an add-on for MS Excel/Word and Open Office which provides templates and enables formal verification of the plan, including a check of the error-prone link column. Moreover, the EDA verification libraries (e.g. Questa Verification IP) supply plans for supported interfaces, which can be directly copied to the user’s document.

2.2 Verification platform

Building the verification platform, the focus should be placed on a generic implementation following the UVM patterns (environment, item execution, etc.). All UVM test specific customization of the verification environment should be applied through the UVM configuration mechanism in the implementation of the given UVM test.

A rapid boost in the testbench development can be obtained with libraries supplied by EDA vendors (e.g. Questa Verification IP [7]). With a few lines of code, the verification engineer can provide a valid stimulus for complex, layered interfaces, such as Ethernet. Moreover, the included scoreboards and coverage monitors will analyse the behaviour of the DUT, indicating, in case of errors, the violated requirements of a given interface standard. It is one of many advantages derived from the use of industrial interfaces, which should be considered at the stage of the DUT design. However, in the case of customised user’s environment-specific protocols the time invested in the development of UVM agents, enclosed in the specific System Verilog packages shared through a common repository, usually saves time in the long-term. Through this collective effort the implemented code matures to a full, ready-to-use, valid verification component. An example repository of such custom interface agents and objects that facilitate Questa Verification IP [7] utilization can be found in [5].

The UVM library provides a factory method pattern inherited from class-based programming. By following it, the user gains the possibility to selectively extend functionality of the developed generic environment at the stage of a test configuration, by substituting one instance by another, more specialised. In this manner, one can add the required coverage models to the basic scoreboard of an interface or define specific constraints to a sequence item in order to cover a corner-case. Moreover, the user can profit from the logging mechanism in the UVM library and gains the possibility to dynamically adjust the verbosity of the testbench, which impacts on the simulation performance.

All signals exchanged with the DUT should be grouped into interfaces. Any hierarchical paths should be avoided, as they always affect the generality of the UVM environment and eventually are a source of cross-compatibility errors. Only the top module linking the simulation environment with the DUT should be hierarchy-aware. In the case of an internal DUT signal access from the verification code, one can use a dedicated interface connected to a required internal component in the top module. Alternatively, one can use a specialized checker defined by the System Verilog standard [10] which externally can be directly bound to the internal sub-modules of the DUT.
The sequence item recording feature of the UVM library allows for an increased readability of the waveforms and indicates the beginning/end time of a transaction. It facilitates debugging of complex, layered interfaces.

System Verilog enables integration of the simulation environment with external software using the Direct Programming Interface (DPI) [10]. It provides the possibility to develop at the same time, at the very early stage, the DUT and the related software. Moreover, the user gains a powerful debugging tool at the border of the hardware and software domains. As an example, this method was used to debug the FPGA-based DAQ system of the CLICpix chip [9]. In this case, the slow control and DAQ software implemented in Python communicates with the hardware over an Ethernet network. For debugging purposes, a simple C++ code opens an Ethernet socket and exchanges Ethernet packets between the socket and the verification environment (over DPI). On the simulation side, a sequence casts the packets to transactions which are provided to an Ethernet agent and then to the DUT. Redirecting a hardware IP address and a port in the Python code to the simple C++ code, one can verify the whole system. The solution is transparent for the software, provided it can handle the longer response time of the verification platform (timeouts).

The DPI can improve debugging capabilities also by exporting transactions to external traffic viewers. As an example, the reference repository [5] includes objects required to export Ethernet items from a verification environment, over DPI, to a C++ code which translates them to PCAP format using the libcap library. The created files can be directly viewed with network protocol analysers such as Wireshark.

Although the generic verification environment is customized by specific tests, a large set of parameters remains common. Thus, a good practice is the implementation of a base UVM test class, a generic test, including a basic configuration of the environment. The specific tests inherit from the generic test, adjust or modify the configuration and provide a set of specialized sequences, e.g. spiTest, chargeInjectionTest, etc..

2.3 Actual verification

The last stage of a systematic verification based on UVM is the simulation. At this step, an issue tracking system (e.g. Jira), improves the communication between the DUT and the verification developers, who are spotting in parallel different bugs and ambiguities of the DUT specification. Usually the verification platform includes a behavioural model of the DUT, all the coverage observers, sequences generating random stimulus, etc. Thus, it contains several times more code than the register-transfer level (RTL) description of the DUT. Thus, the first runs typically reveal bugs in the verification code itself.

In most cases the simulation tool is a single thread software. Hence, once the verification environment is stable, a significant time reduction in achieving the full coverage can be obtained by running simultaneously shorter tests launched with different randomisation seeds.

EDA vendors provide tools which support users in spawning parallel simulations on a single machine or on a grid farm. As an example, Questa Verification Manager from Mentor Graphics is presented in figure 4. The tool reduces and improves the required maintenance. All actions

\[\text{\textsuperscript{1}}\] For large designs, having a special license, one can utilize a partitioning approach and parallel simulation. However, it creates complications for grid farms as the resource utilization becomes unpredictable.
Figure 4. Snapshot of the Questa Verification Manager.

(parallel/sequential compilation, optimization, simulation) and configuration (randomization seeds, file paths, etc.) are controlled by a single XML file. The interface of the tool informs the user about progress of the tests and their status (pass/error). Moreover, one can open a dedicated test in the simulator in order to view waveforms and debug the revealed problems.

Figure 5. The overall verification coverage.

At this stage of the verification process, the aim is to reach total coverage without errors (DUT bugs). The verification manager imports the verification plan and reports the achieved coverage metrics (figure 5). It allows one to estimate the overall verification progress. If needed, one can add specialized sequences in order to quickly reach the corner cases of the verification. However, achievement of 100% functional coverage does not necessarily mean a full verification of the DUT functionality. The verification is as detailed as the test plan specified at the beginning of the process. A good cross-check of the simulation is the code coverage — if one reaches 100% functional coverage with 50% code coverage, the test plan may be incomplete and requires updates. Similarly, 100% code coverage does not mean that all the possible concurrent interactions of behaviour within, or between multiple design blocks occurred during the simulation. Hence, to get a complete picture of the DUT verification progress, one often needs multiple metrics.

After the successful verification of the RTL design, especially in case of ASIC projects, the actual verification should be repeated, substituting the RTL description with a back annotated netlist which provides a more realistic DUT description including internal delays.
3 Summary

Using the UVM methodology and taking advantage of the tools provided by EDA vendors, one can efficiently build a complex and versatile testbench utilizing various interfaces (Ethernet, I2C, SPI, custom) to stimulate the DUT. Moreover, the described well-established verification patterns allow an exhaustive system verification and identification of difficult-to-track design flaws.

References

[1] Universal Verification Methodology (UVM) 1.2 User’s Guide, http://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf
[2] Accellera Systems Initiative, http://accellera.org/.
[3] A. Fiergolski, Hardware implementation of the track identification algorithm in the Scalable Readout System for the TOTEM experiment, CERN-ACC-2015-0050.
[4] Mentor Graphics, Verification Academy — cookbook, August 2013.
[5] HDL Verification Library, https://gitlab.cern.ch/CLICdp/HDLVerificationLibrary.
[6] Questa Advanced Simulator, https://www.mentor.com/products/fv/questa/.
[7] Questa Verification IP, https://www.mentor.com/products/fv/verification-ip.
[8] EUROPRACTICE Software Service, http://www.europractice.stfc.ac.uk/welcome.html.
[9] P. Valerio, A. Nascetti and R. Ballabriga, Electronic Systems for Radiation Detection in Space and High Energy Physics Applications, CERN-THESIS-2013-156.
[10] IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language, in IEEE Std 1800-2012, revision of IEEE Std 1800-2009, February 21, 2013.