Making it Easy to Deploy the UVM

by Dr. Christoph Sühnel, frobas GmbH

ABSTRACT

The Universal Verification Methodology (UVM) is becoming the dominant approach for the verification of large digital designs. However, new users often express concern about the effort required to generate a complete and useful UVM testbench. But the practical experience collected in numerous OVM and UVM projects during the last few years shows a different view. The UVM is a very suitable methodology for any kind of design and implementation, i.e. ASIC and FPGA due to the availability of the UVM library and the well-defined testbench structure. This allows the automation of essential steps in employing the UVM.

This article describes an UVM approach reducing testbench implementation effort, guaranteeing an early success and streamlining the processing of the test results. Depending on the number of functional interfaces and the design complexity up to 6 weeks of implementation effort or even more can be saved. A runnable UVM testbench will be handed over to the verification team at the very beginning of the project. The verification engineers have to implement only the corresponding drivers, monitors and functional coverage models. Later on the scoreboards needed have to be architected and implemented.

The results of the test execution are processed by scripts giving the project manager a continuous figure about the project status and progress.

INTRODUCTION

The Universal Verification Methodology (UVM) is an Accellera standard. It was built on the existing OVM (Open Verification Methodology) library (code base) and contributions from the Verification Methodology Manual (VMM).

The key technologies employed in the UVM are Object-Oriented Programming (OOP) and Transaction-Level-Modeling (TLM). They give this approach a high flexibility, raising the abstraction level and enable plug & play reuse. But they are completely new to all hardware designers and most verification engineers. This might be the main reason why UVM is considered to be useful for large ASIC designs only. But the industrial experience shows this is not true. The UVM provides best prerequisites to be employed independently of the design complexity and their silicon implementation. It is not limited to ASICs, but also very useful for FPGAs. The question is how to make the employment of UVM most effective.

The UVM library can be considered as a big, but well-structured construction set for testbench generation as well as sequence and test implementation. The UVM User Guide and additional documents like the UVM Cookbook provide rules, recommendations and examples how to use the single components of the UVM library best.

Due to the structured approach of the UVM library the main architecture of an UVM testbench looks always the same. It is based on agents or sub-environments for each functional interface of the design-under-test (DUT). But each agent has the same structure. It consists of 4 blocks: a sequencer, a driver, a monitor and a coverage collector. The complete testbench consists of several agents or sub-environments, instantiated in a top-environment. This uniformity allows the automation of the generation of a complete UVM framework. This article describes an approach speeding-up the testbench generation and making the project startup easy.
UVM TESTBENCH ARCHITECTURE

The common UVM testbench architecture is shown in hierarchical form in Figure 1 to Figure 3. It consists of a bunch of UVC (Universal Verification Components) environments, one for each functional interface of the DUT. Each sub-environment consists of 2 objects, the agent and a corresponding configuration object. Each agent has 5 objects, driver and sequencer, monitor and coverage and the configuration. The virtual interface (vif) is handled as a configuration object. The overview is shown in Figure 1.

The top environment summarizes the sub-environment and includes additional and optional units like a scoreboard and a register model. For details see Figure 2. Additionally you can find application-specific blocks like scoreboards and register units.

The test object instantiates all configuration objects as well as the toplevel environment. It passes configuration objects like the virtual interfaces through the UVM hierarchy to the blocks where they are needed. Especially virtual interfaces which can be retrieved by drivers and monitors. The overview is shown in Figure 3.

The testbench architecture described here is nothing special, but is rather generic. This structure is commonly used in industrial verification projects employing UVM. But it might give you an impression about the effort to be spent for implementing this generic UVM testbench.
structure. It is a boring task. During implementation lots of SystemVerilog syntax errors and mistakes in implementing transaction-level-modeling could happen. This discourages verification engineers and makes it difficult for project managers to develop an accurate estimate of the effort and time to generate a UVM testbench. This high barrier to adopting the UVM often prevents especially FPGA projects and inexperienced engineers from deploying UVM. Lowering this barrier will make UVM more acceptable to a wider range of projects and companies.

This article describes an approach helping to apply UVM for any kind of projects (large or small) and implementations (ASIC or FPGA). It has been proven in numerous successful projects and was adopted by a lot of companies. Verification engineers as well as project managers were happy to see how simple UVM can be.

**AUTOMATION OF THE GENERATION OF A UVM ENVIRONMENT**

**Extracting setup data**

The target object for verification is the design-under-test (DUT). A testbench has to drive input ports of a design and observe output ports. Architecting a testbench we have to make sure all ports of the DUT are connected correctly to the verification environment. The best way is to structure the ports of the DUT with respect to the functionality behind. In this way we get groups of ports with respect to functionality. For the SPI functionality this means the ports sclk, cs, miso and mosi are belonging to a functional interface. Doing this for all ports we get groups of functional interfaces. Each functional interface will be assigned to a sub-environment in the UVM. For the UVM testbench generation these signals are collected in a SystemVerilog interface construct. Figure 4, below left, shows the port-list of a design in Verilog. The DUT possess 2 functional interfaces, a SPI and an APB functional interface.

The corresponding SystemVerilog interface construct is shown in Figure 5 below. The names in the port-list do not have to be identical to the names in the interface construct.

 Additionally data to be transmitted to the UVM testbench are relevant for verification and sequence generation. These data are addresses or data written to registers, data to be transmitted in a communication system. Such data are relevant for verifying a design and they are never considered as control signals. Data like this have to be generated by sequences and they will be defined in the sequence item description and transferred on the transaction level. For the SPI block such data might be the data to be transmitted, the length of the data packet and any additional information. Figure 6, above right, shows a possible implementation of data to be generated in the UVM testbench.

The description above gives you an impression what kind of data are of interest and which effort have to be spent to get key data for the generation of a UVM testbench. You might have seen there is no special knowledge needed to extract the setup data out from the DUT and its specification.
The UVM framework generator is a Perl-script. It takes care to generate the complete UVM base architecture. But setup information has to be provided in the right format. For each functional interface a separate setup-file has to be created, based on the information explained above. For the SPI functional interface this setup information looks like what is shown in Figure 7 below.

The content description is as follows:

**uvc_Name**: name of the UVC to be generated;

**uvc_item**: name of the sequence item;

**uvc_var**: variable to appear in the sequence item; this is commonly a whole list;

**uvc_if**: name of the SystemVerilog interface construct;

**uvc_port**: port to appear in the interface; this is commonly a whole list.

Most companies want to have in each file a header with information about the project, the author and his contacts. For this data a common setup file is available, as shown in Figure 8 below.

**Figure 6: Data in the SPI sequence item**

**Figure 7: SPI setup data**

**Outcome of the UVM Framework Generator**

The UVM framework generator creates the testbench, a few example sequences, including a virtual sequence approach for coordinating sequences and tests. The corresponding data structure as shown in Figure 9 below.
spi_tb is the project root directory. The main sub-directories are:

- dut: contains the DUT description or links to it;
- sim: has a compile and simulation script inside, generated by the framework generator;
- tb: contains the subdirectories of the whole UVM testbench.

Figure 9 shows a complete UVM testbench data structure for 2 UVCs (spi, apb) with a register package inside (apb_rgm).

spi_top contains the toplevel environment, integrating the spi and abp UVC as well as the register package.

spi_top_tb is the directory with the toplevel module of the UVM environment and spi_top_test contains all the test necessary for verification.

To get an impression what files are in a UVC directory look at Figure 10. This list of files you can find in the current example in the directories spi, apb and spi_top.

But the DUT ports are not driven because the protocols for the functional interfaces are still not there. They have to be implemented next to stimulate the design.

The flow for employing the UVM Framework Generator is shown in Figure 11 below. It summarizes the steps described above.

**Benefits of the UVM Framework Generator**

The application of the UVM Framework Generator improves the verification productivity and lowers the entry barrier to the UVM dramatically. This might encourage a wider range of potential users to adopt the UVM as their preferred verification methodology.

In detail the benefits of the UVM generator can be summarized as follows:

- takes over formal steps in the testbench, sequence and test creation;
- gives the verification engineer more time for the creative steps in verification;
- helps to avoid mistakes and breakthroughs;
- helps to educate in SystemVerilog and UVM usage;
- guarantees the uniformity of the testbench in additional projects and simplifies the maintenance of a UVM project;
- speeds up the verification process;
- guarantees high quality of the verification;
- simplifies the entry to the UVM;
- motivates verification engineers by avoiding boring errors during implementation;
- data base structure supports reuse.

The industrial usage of the UVM Framework Generator shows all projects executed were very successful. The customer’s management, as well as verification engineers, are happy with these projects.

The savings for a medium size DUT were estimated with 6 weeks or even more of implementation time. Employing an UVM coach helps to perform an effective additional education and knowledge transfer of the customer’s staff, helps to be on target in the project.
AUTOMATING THE VERIFICATION MANAGEMENT PROCESS

The objective of the verification management is to summarize verification results and analyze against the verification plan. These results give an accurate figure of the actual status of the verification process. And it helps the test writer to identify holes in the verification process and gives input for additional sequences and tests.

Prerequisites for the verification management are a verification plan, readable by the simulator and simulation results in a special format. Questa uses the Unified-Coverage-Data-Base (UCDB)-format for storing verification results like coverage (functional and code), assertions and test results. The verification plan uses source and entry format like Microsoft Word or Excel or Framemaker. For further processing the plan has to be stored in the intermediate xml-format. This can be read in by Questa and stored also in the UCDB-format.

Furthermore Questa has useful commands for processing verification data and generating verification reports. These commands were applied in a TCL-script which can be run on a daily basis to rank, merge and analyze the verification results and generate verification reports. A verification report as shown in Figure 12 will be created.

Figure 12: Coverage Report
Ranking of tests indicates how much each test contributes to the total coverage. This information is of interest to identify redundant tests. Furthermore, this helps to select tests for gate level simulation, where not all tests will be executed. A cutout of the ranking report is shown in Figure 13. The middle number column shows the total coverage numbers and the right one the increments reached with this specific test. The selected contributes to the functional coverage (covergroups/coverpoints) and to special code coverage items (branch, statement, and toggle). It might not be a good candidate for gate level testing.

**Figure 13: Ranking Report**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Items</th>
<th>Covered%</th>
<th>Inc%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cover Groups/Points</td>
<td>6/36</td>
<td>48.87</td>
<td>18.33</td>
</tr>
<tr>
<td>CoverDirectives</td>
<td>31</td>
<td>51.61</td>
<td>0.00</td>
</tr>
<tr>
<td>Statements</td>
<td>2792</td>
<td>80.80</td>
<td>0.71</td>
</tr>
<tr>
<td>Branches</td>
<td>1986</td>
<td>68.87</td>
<td>0.90</td>
</tr>
<tr>
<td>ToggleNodes</td>
<td>4581</td>
<td>53.96</td>
<td>1.20</td>
</tr>
<tr>
<td>States</td>
<td>17</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Transitions</td>
<td>45</td>
<td>84.44</td>
<td>0.00</td>
</tr>
<tr>
<td>AssertPasses</td>
<td>10</td>
<td>100.00</td>
<td>0.00</td>
</tr>
<tr>
<td>FecExpressions</td>
<td>440</td>
<td>64.31</td>
<td>0.00</td>
</tr>
<tr>
<td>FecConditions</td>
<td>961</td>
<td>25.91</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**CONCLUSION**

Reading articles and books, investigating code examples the UVM seems to be highly complicated. This article wants to change this impression. By considering the UVM library as a huge construction set with well-designed pieces and by automating time-consuming formal steps the entry barrier will be lowered and the effort for verification reduced by increasing the verification quality. Running a verification management script on a daily basis will display the verification progress and give verification engineers and project managers an excellent figure about the verification progress and status.