Simplified UVM for FPGA Reliability
UVM for “Sufficient Elemental Analysis” in DO-254 Flows

by Shashi Bhutada, Mentor Graphics

INTRODUCTION
DO-254 and other safety critical applications require meticulous initial requirements capture followed by accurate functional verification. “Elemental Analysis” in DO-254 refers to the verification completeness to ensure that all ‘elements’ of a design are actually exercised in the pre-planned testing. Code Coverage is good for checking if implementation code has been tested, but cannot guarantee functional accuracy. Currently, functional accuracy is guaranteed using pre-planned directed tests, auditing the test code and auditing the log files. This is not scalable as designs get complex. In this article we will look at using SystemVerilog syntax to concisely describe the functional coverage in the context of accurate “elemental analysis”.

“Safety Specific Verification Analysis” as called out in DO-254 Appendix B addresses the need to check not only intended-function requirements, but also anomalous behaviors. It is left up to the applicant to propose a method to sufficiently stimulate the “element” to expose any anomalous behavior. In this article we shall focus on this “Safety Specific Verification Analysis” or “Sufficient Elemental Analysis”. We will be using UVM to describe the stimulus space to look for anomalous behavior.

The article will also highlight items to use for auditing a flow from DER’s point of view.

VERIFICATION COMPLETION CRITERIA
The key stakeholders for a FPGA (or ASIC) Verification include DER, Managers and Engineers. The DER is responsible to audit the flow with the focus on design safety. The managers are trying to meet the customer requirements with shorter design schedules and optimal resources. The designers want to accurately capture the design intent using the tools provided.

The purpose of verification is considered to be a “supporting process” in a DO-254 program. It is not a specific phase of development, but rather occurs throughout the design flow from the earliest models to the final testing of the component in the system. The primary objective is to ensure that a design performs the function specified by its requirements and that it satisfies agreed-upon completion criteria. Safety critical DAL A and B devices require that the verification be carried out independent of the design.

“Elemental Analysis” in DO-254 refers to the verification completeness to ensure that all ‘elements’ of a design are actually exercised in the pre-planned testing. An ‘element’ is the smallest design item that an engineer uses to create the design. In the context of VHDL/Verilog language based design of FPGA/ASIC devices this can vary from a statement to a conditional block, or some other larger structure such as a reusable internal/external IP, or a reusable bus interface protocol.

Code Coverage is a good metric for determining if the design implementation code statements, conditional blocks or FSM coding structures are exercised by simulation. But Code Coverage will not assure the functional accuracy, integrity of a reusable IP block or a bus protocol. This leads to writing too many redundant tests and wasted simulation cycles. See Figure 1. Did the higher Code Coverage mean the design is functionally accurate? Did the manual audit miss functionality? Did the larger design state space get exercised to look for any anomalous behavior? Is there unnecessary testing wasting simulation cycles?

When planning for sufficient elemental analysis at every stage of the design flow one needs to confirm things identified in Table 1.

![Figure 1: Types of Tests](image-url)
DERs would like a tool that showed the spec in an executable format for stimulus and response. This can reduce the time currently spent on manual audit. Managers would like to eliminate unnecessary redundant waste of simulation cycles and designer’s time. And designers would like to use the latest methodology to make their delivery more reliable and robust.

The testing needs to be pre-planned to cover the required modes and configurations, but it also needs to make sure other unused configurations and modes have predetermined safe behavior. SystemVerilog can be used to capture the external stimulus/response spec in a readable and executable format. To guide the stimulus we can use Functional Coverage. It is also important to have a means of sufficiently testing the larger design state using a vast range of scenarios beyond the pre-planned “directed” necessary tests. The SystemVerilog constructs are made much more consistent to use via UVM techniques.

**UVM (UNIVERSAL VERIFICATION METHODOLOGY)**

UVM is an open-source library of SystemVerilog language-based source code maintained by Accellera.org. It is developed by a conglomeration of companies and independent developers. It is based upon various precursors of the technologies including proprietary technologies developed and deployed for complex hardware verification over the decades. It is supported by all the major simulation tool vendors and is widely exercised by thousands of projects across the industry. There is a large ecosystem building around UVM to provide code development editors, verification IP, debuggers, and UVM-conversant college graduates.

UVM testbench has the similar goals of applying test vectors and measuring the response as shown in Figure 1.1.

### Table 1: Verification Completion Criteria

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Stimulus</th>
<th>Functional Metrics</th>
<th>Questions to answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-planned tests</td>
<td>Apply specific stimulus</td>
<td>Spec Coverage</td>
<td>Test delivered</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Functionality</td>
</tr>
<tr>
<td>Robustness within a given range</td>
<td>Apply wide range of stimulus</td>
<td>Range Coverage</td>
<td>Test accuracy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>in a range of test cases</td>
</tr>
<tr>
<td>Robustness outside the expected range</td>
<td>Apply range of error-stimulus</td>
<td>Error-case Coverage</td>
<td>Test design tolerance</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>to errors</td>
</tr>
<tr>
<td>Corner cases</td>
<td>Apply corner—tests beyond</td>
<td>Corners Coverage</td>
<td>Exercise corner behavior</td>
</tr>
</tbody>
</table>

UVM testbench has the similar goals of applying test vectors and measuring the response as shown in Figure 1.1.
The major difference with UVM is that the structure of UVM testbench is component-based and uses improved object-oriented syntaxes added in the SystemVerilog language. This allows plug-n-play and reuse as shown in Figure 1.2.

Let’s look at some of the key aspects of the UVM testbench using a simple example. We will also highlight how these aspects help a DO-254 type of project.

The design we chose as an example is a Floating-point Unit (FPU) as shown in Figure 2, available to download from OpenCores.org. As seen in the figure, the total number of input bits including the operands and operations is 69-bits. From a testing perspective we cannot possibly test all the 69-bit input data bit combinations \(2^{69}\). The IP comes with a VHDL testbench. It uses 100,000 test cases for each operation and rounding mode amounting to about two million stimulus vectors. The input and expected vectors are provided via a text file. It was followed by hardware testing. But is this IP fully verified?

We will approach the verification using UVM. The primary verification requirements for the FPU are as follows:

- Verify all the five operations and four rounding modes – 20 combinations in all
- Verify 32-bit IEEE-754 compliant floating point data inputs
- Verify that exceptions get flagged correctly

UVM testbench development starts with the interface declaration, which is just a collection of DUT pins as shown in Figure 4. This collection of pins is very similar to a module declaration in Verilog. It can also include the assertions to verify timing activity around the interface, such as the number of pipeline delays needed to complete each operation. In this case we want to make sure of the following items:

- Add/Sub operations takes seven cycles
- Multiply operation takes 12 cycles
- Division/square-root operation takes 35 cycles

SystemVerilog assertion syntax allows clock-based timing checks plus data capture at various clock points. This interface can be bound to a VHDL DUT and VHDL testbench to add the notion of functional coverage without modifying the code. But our focus here is to explore UVM.
The next step is to build capture the stimulus description. This is done by defining transactions. The transaction contains two operands, operation and rounding mode. Transactions are defined at higher level and never talk about control signals or the timing of the bus interaction. The operands are 32-bit floating point numbers. To limit the testcases plus perform exhaustive verification we break down the 32-bit space into 12 ranges as shown below.

1. Positive and Negative zero
2. Positive and Negative Denormalized Real Number with exponent=0
3. Positive and Negative Normalized Real Number with exponent > 0
4. Positive and Negative Infinity
5. Positive and Negative Quiet NaN (Not a Number)
6. Positive and Negative Signaling NaN

This reduces the total stimulus combination to be 12 types of operand A, 12 types of operand B, four operations and five rounding modes. This amounts to a total of 2880 stimulus combinations. We are in essence building stimulus knobs and also response meters as shown in Figure 5.1.

Transactions are defined in UVM using SystemVerilog “class” syntax. This is very similar to VHDL record to encapsulate data but SystemVerilog “class” further allows extension of a type. Each class can have attributes. SystemVerilog “constraints” allow defining the valid range of values for those attributes. See Figure 5.2 on the following page for details on the operand “class” declaration. The attributes have a range of suitable values defined inside “constraints”. It currently only shows Normalized and Denormalized type of operands. This file can be made to exactly align with the stimulus specs and can be used for auditing the ranges defined. Other ranges can be declared using similar syntax.

The UVM Sequence Item or the transaction that will stimulate the DUT is a collection of the operands and operations. It will be used to build the sequences or the testcases as shown in Figure 5.3 on the following page. Notice how the fpu_request declares the class as an extension of uvm_sequence_item, which is part of the UVM library and comes with various utilities and extensible functionality. Also, notice how the fpu_response extends the request transaction and inherits the fpu_request attributes plus adds its own attributes. Inheritance is one of the key features that helps reuse.

The UVM Sequences are a string of transactions. IP-provided VHDL testbench used a C-program generated equivalent Knobs and Meters.
test vectors to generate stimulus. UVM uses constraint randomization as shown in Figure 6.

Inheritance allows a parent to share a functionality or a method to its child. The “body()” function is the main built-in method that generates the transactions. This is overridden by the user code which defines the order of generation of sequence items. This can prove as a better way to document the testcases for DO-254 and can prove as a better artifact than two million VHDL test vectors.

Sequences use the knobs we have built earlier. The randomize() function invokes the simulator-built-in constraint solver that solves all the constraints to pick a solution. The “randomize() with { … }” allows further tightening the constraints. If there are any constraint conflicts the function will return “0” and the “assert” around the randomize() call will catch the failure. At the end of the randomize() call the “rand” attributes within the class will have a value picked from the constraints-solved solution space. Each new seed specified picks a new testcase from the space.

The UVM structural blocks that drive the transactions onto the interface (onto the DUT) and independently monitor the bus activity are shown in Figure 7.

The UVM Sequencer reads the sequences we have created. It is the simplest component created as shown in Figure 7.
The UVM Driver reads the sequence items generated by the Sequencer (via the chosen sequence). The driver accesses the FPU pins via “virtual interface” and provides the cycle-accurate bus pin wiggling on the DUT. The “run_phase” is the built-in uvm_component “virtual method” that gets invoked or orchestrated by the UVM base code and is customized by the user as shown here. Notice also the use of “forever” loop inside “run_phase()” task that extracts the data generated by the sequence via the built-in “seq_item_port.get(m_request)” call. The component’s run_phase() tasks are called as the threads that get launched simultaneously and typically coded to run forever. Sequences typically control the simulation runtime until no items are generated and all threads are killed by UVM base code.

The UVM Monitor independently monitors for requests and responses for broadcasting the data to the rest of the testbench. Notice the analysis_port.write() function invocation which is the way to broadcast the observed transaction. Here is where one could also integrate a way to print out stimulus vectors applied and response seen in a file for DER Audit purposes. We will see next how the listeners or subscribers respond to this call.

We are skipping the details on how the UVM components are hooked up and or the details on UVM Agents, UVM Environment and UVM Test. The goal here is to architect the transaction to drive proper stimulus, see it as a documentation artifact. One can also review the bus interface, the corresponding driving/monitoring functionality, if needed.

The UVM Subscriber component, as shown in Figure 8 and Figure 9 is a listener to the monitor’s analysis_port.write() function call that broadcasts the observed response transaction. Subscribers receive...
the transaction by implementing “write()” callback function. More than one subscriber can be added to listen to a single
monitor analysis port.

The UVM Scoreboard listens to the stimulus and responses. It takes the applied stimulus and predicts the expected output. The prediction algorithm can be C. Scoreboard also compares the expected response with the actual response. See Figure 8. Also, notice the consistent way of message logging.

We can integrate coverage collectors as listeners or subscribers as well. For example, one of the checks required is to check sequential FPU operations – such as ADD followed by SUB, ADD followed by MUL, etc. We have five total operations. This makes it 25 possible sequential operation combinations. This is done via SystemVerilog covergroup syntax that allows transition coverage syntax. This is a single line metrics collector. See Figure 8 which shows the CoverGroup Functional Coverage used to capture valid ranges of data covered plus sequential operations transition coverage.

This covers an introduction to some of the essential aspects of UVM. There are other structural aspects of the UVM testbench such as an agent, an environment, and a test. For further details please look at the UVM Cookbook or VerificationAcademy.com.

See Table 2 below for a recommendation on the specific artifacts to be used by DER for a review of a UVM testbench.

**SUMMARY OBSERVATIONS**

Code Coverage is one of the recommended and required tools in DO-254 flows. The verification items identified during the course of the example description earlier cannot be covered via Code Coverage, the metric that helps measure these types of verification items is called Functional Coverage. The traditional way to test this would be to write specific testcases for each of the functional scenarios. The result is printed out in a log file which

<table>
<thead>
<tr>
<th>Tests to Audit</th>
<th>Review Source Code</th>
<th>Review Results</th>
<th>Review Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-planned tests</td>
<td>Constraints</td>
<td>Stimulus Vectors</td>
<td>Code Coverage</td>
</tr>
<tr>
<td>Robustness test (within range)</td>
<td>Sequences</td>
<td>Response Vectors</td>
<td>Functional Coverage</td>
</tr>
<tr>
<td>Robustness test (outside the range)</td>
<td>Functional Accuracy Coverage</td>
<td>Run log files</td>
<td>Robustness Coverage</td>
</tr>
<tr>
<td>Corner tests</td>
<td>Protocols in Driver/Monitor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 2: Important files from DER Audit point of view*
is meticulously reviewed. This does involve extra code and extra time to review. Functional Coverage is typically measured via assertions and covergroup syntax in SystemVerilog. Assertions keep track of temporal timing diagram based activity. Covergroups can keep track of data items observed such as types of operations or which of the 12 buckets of the data the operand has hit. We can use CoverGroup and Assertions directly along with VHDL testbenches.

When the assertions and covergroups were applied to VHDL testbenches it was noticed that even 2000 random vectors gave the same coverage as two million vectors. This highlights the fact that more tests doesn’t mean new items are being tested – it merely is repeating redundant tests, which in turn means wasted simulation cycles and wasted productivity. In our measurement we also saw that only a small set of the 2880 combinations was hit by the stimulus which means the stimulus space was not explored intelligently. Adding Functional Coverage provides the visibility on verification efficiency.

When we applied Functional Coverage using assertions and covergroups to the existing VHDL testbench (without modifications or adding UVM) we noticed the following:

- 2000 and up to two million test vectors only achieved about 75% coverage
- Out of the optimal 2880 input vector combinations only about 5% were tested
- Out of 25 sequential operations only 40% were tested
- We noticed three out of five pipeline delay properties fired highlighting bugs in the design or the spec

UVM provides a better structure to deploy SystemVerilog Assertions and CoverGroups. We noticed that with UVM testbenches the verification coverage was achieved faster. This allowed us to do further exploration leading to finding a bug in the design. Here are the observations with UVM:

- We wrote five sequences ran them with two different seeds amounting to 10 tests in total
- We got about 97% functional coverage with 10 tests and 1/10th amount of time
- We covered 100% of the sequential combinations of the operations
- We found a bug with square root operation

In conclusion, two million testcases in VHDL testbench sounded good but SystemVerilog Functional Coverage provided quantification. This article was intended to cover an introduction to some of the essential aspects of UVM, cover how it can help DER auditing process plus how the design/management team can benefit. The article should also provide some hint on the mindset needed for deploying UVM.

REFERENCES:

1. UVM: http://accellera.org/downloads/standards/uvm
2. Floating-point Unit VHDL Design: http://opencores.org/project,fpu100
3. Floating-point Standard: IEEE 754
4. Effective Verification for DO-254, David Landoll, MAPLD 2008
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

25 Video Courses Available Covering
• Formal Verification
• Intelligent Testbench Automation
• Metrics in SoC Verification
• Verification Planning
• Introductory, Basic, and Advanced UVM
• Assertion-Based Verification
• FPGA Verification
• Testbench Acceleration
• PowerAware Verification
• Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 6500 topics
UVM Connect and UVM Express Kits

www.verificationacademy.com