2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

How to Verify Complex FPGA Designs for Free

Sebastian Dreßler, Nikos Anastasiadis, Thomas Richter, Swarm64 AS, Berlin, Germany
(sebastian@swarm64.com, nikos@swarm64.com, thomas@swarm64.com)

Abstract—The correct verification of FPGA-based designs is a key to successful product development. However, startups and young companies also have to meet budgetary limits. This limits the selection of tools to do verification. Swarm64 is a startup designing a complex FPGA-based hardware and software solution for database acceleration. We chose C++ to successfully verify our design. This paper shows our approach and endorses "verification for free" with the help of Open Source Software.

Keywords—c++, complex designs, coverage, fpga, verification

I. Application

Swarm64 develops a FPGA-based hardware and software solution for database acceleration. The hardware part of our product is a PCIe card with customized processing units, a cache hierarchy and interfaces to SSDs. All these modules are written in Verilog. Verification of our RTL code is one key to successfully develop our product. However, as a startup we likewise experience more financial constraints than large multinational companies. As a result, we had to be creative in achieving industry standard verification quality with non-standard tools. One of the founders of the company[1] had been part of the team that developed the ARM Mali and thus extensive experience with industry standard verification of complex hardware. Being initially unable to afford tools for standard SystemVerilog and the accompanied UVM or similar verification approaches, Swarm64 developed its own verification approach based on an Open Source toolchain. It is still in use today alongside industry grade verification tools and thus is presented in this paper as a complementary verification approach for complex projects or as a solution for low-budget projects.

To meet our quality and stability goals while considering the aforementioned restrictions, we chose to validate our design by applying constrained random stimuli and scoreboards written in C++ combined with the Open Source Software *Verilator* [1] and [2]. This software transforms Verilog source code to a C++ model. Thus, our approach is as follows. We first transform the device under test (DUT) to its C++ model. Then, we connect the testing logic to the model. Eventually, both are compiled to an executable which then performs the simulation. To connect the C++ test logic with the "verilated" model, we use *SystemC* since *Verilator* supports its data types and also its simulation kernel.

The aforementioned test logic consists of a combination of C++ classes provided by our in-house verification framework. Its design was inspired by UVM and provides two generic building blocks. These are i) verification components (VCs), and ii) communication channels (CCs) to easily interconnect VCs. Modules based on VCs are: constraint random stimuli generators, interface drivers, checkers and scoreboards. CCs can for instance perform type conversion, introduce delays into the system, and reorder commands.

Based on the simplistic principle of interconnecting processors with queues and by using modern C++ language features such as templates and lambda functions, it is achievable to implement complex tests with greatly reduced effort. In fact, we are able to simulate our complete system in a reasonable amount of time, yet still replay target log dumps with our simulator and design tests to validate every module in the system.

We measure the quality of this method by using line coverage and signal toggle coverage metrics. That is, during simulation, the toggled signals are measured. As a post-processing step, this information is combined with the source code showing us which lines of code are covered by the simulation. By tweaking simulation

Acknowledgements to Luc Vlaming and Michael Zimmer for valuable discussions.
[1] Eivind Liland

parameters or by designing new tests, we increase the coverage and thus increase the quality of our verification. We note, that by relying on these metrics, our approach should be used for FPGA-based designs only.

After citing related works, this paper will first cover the implementation of the testing environment with the proposed components and workflow. Then the *SystemC* framework, as a link between the hardware and C++ aspects, is introduced. The example test described in detail illustrates how the concepts of this paper integrate together in practice. Finally, the results we measured with respect to the quality of our verification method and its performance are described. We conclude with discussing the relevance of our paper in a broader context.

II. Related Work

Our work relies on the Open Source Software *Verilator* [1], which performs the necessary source-to-source transformation from Verilog to C++. It also provides the simulation runtime. Otherwise, it would have been required to use a (commercial) simulator for our initial verification process.

Furthermore, [3] and [4] likewise propose a C/C++ based workflow. However, their approaches differ in the extend of *SystemC* usage. While we rely on *SystemC* only for connecting the interface from the test environment to the DUT, they use *SystemC* as basis for simulation. Especially when it comes to the simulation of the DUT, a simulator capable of combining RTL and *SystemC* is required.

III. Implementation of the Testing Environment

Figure 1 depicts the outline of the test environment. In general, it consists of three major building blocks, which we explain briefly in the following paragraphs.

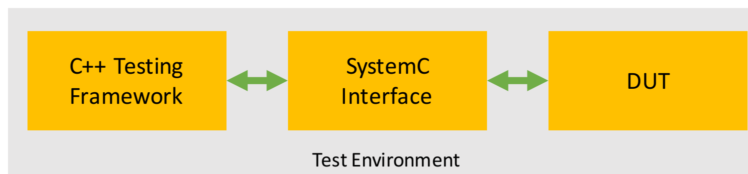

Figure 1. Testing Framework Overview

First, there is the *C++ Testing Framework* (TF), which provides everything needed to execute the test. That is, it provides the aforementioned VCs and CCs. Furthermore, it also implements the logic to drive these components. By driving we mean that the TF provides the main run loop which triggers all registered VCs. We will explain this functionality in detail in the next section.

The second building block is the *SystemC Interface* (SCIF). This component is responsible for interconnecting the *Device Under Test* (DUT) and TF to drive / receive data and commands to / from the DUT.

The DUT represents the last building block. This C++ code generated by *Verilator* which directly models the RTL-code being tested. The *Verilator* tool performs a source-to-source transformation from Verilog to C++ and thus resembles the core process of our verification process to obtain an equivalent C++ model from RTL source code. *Verilator* provides us the capability of not using a simulation tool but instead obtaining an executable which is standalone and can be run without restriction. Due to this fact we are also able to run multiple simulations in parallel, which reduces overall simulation time thus helping to find more bugs in less time.

We now describe the TF and SCIF in detail. For that, we first explain the purpose of VCs and CCs followed by an explanation of the test running logic.

A. *Detailed Description of the C++ Testing Environment*

1) Conceptual view on verification components and communication channels

The core concept of the TF is to provide simple to use building blocks used for performing specific tasks with respect to verification. These building blocks are referred to as VCs. A VC can be seen as small unit which works on command packets. A command packet is a data packet related to the DUT. For example, assume a DUT with a request channel. On this request channel the DUT expects a combination of *opcode* and *address*. Within the TF, this combination is represented by a command packet. Performable actions by the VC on these are for instance: generation, updating, checking (e.g. responses from the DUT) and routing.

However, VCs alone are not sufficient for building a test harness to drive a DUT. To finally enable this, the TF furthermore provides CCs to interconnect VCs. With this interconnect capability, it becomes possible to pass *command packets* from one VC to another VC. This principle is also visible in Figure 2. However, CCs serve also a second purpose. That is, they enable performing actions with respect to communication on the *command packets*. These actions are for example: conversion, delay and reordering. It is furthermore important to note that objects are *moved* between VCs. In C++ this ensures that there is only a single copy of an object at a time.

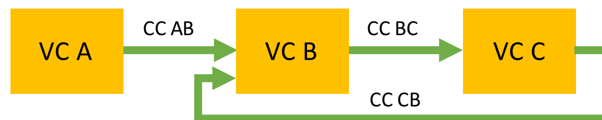

Figure 2. Interconnect example of Verification Components (VCs) and Communication Components (CCs).

2) Implementation and examples of verification components

With respect to C++, a VC is a class which inherits from a pure virtual base class. That is, the base class provides an interface definition but cannot be instantiated on its own. The inherited class however has to provide the required methods otherwise it would not compile.

Using a common base class for all VCs has the benefit that all VCs are storable within a single array. This in turn is useful when it comes to the test running logic because it then can iterate over all available VCs in the whole test and interact with them.

The required methods being implemented by each subclass are *process* and *done*. The main objective of *process* is to work with *command packets*. *Done* in turn provides information whether the VC is finished with all its operations. VCs support up to two types of *command packet* at a time. That is a VC may support a different input and output type. These types are provided as C++ template parameters and thus are checked during compile time which ensures type safety. Another two template parameters specify how many input and output ports are there. These ports are again tied to the input and output types.

For example, if the VC is a command generator, it would provide zero inputs and a single output port with the appropriate *command packet type* being attached to it. The *process* method would implement logic to create a constraint randomized *command package* which is then put into the output port. The *done* method would return *true* if all *command packages* were generated. The amount of these is configurable per test.

Another VC example is a router which has access to *N* inputs and *M* outputs, routing with respect to some algorithm. Now, the VC has N input ports and M output ports of which all share the same type. The *process* method has a function argument passed in during construction pointing to the routing algorithm. This function would take the port-number of the input and produce the port-number of the output. Having this information, the routing VC is able to move the incoming *command package* on any input port to the correct output port where it

can be picked up from another VC. In this particular implementation, *done* would always return *true*, because there is no knowledge on how many items are to be routed.

3) Implementation and examples of communication components

CCs are implemented in a similar fashion as VCs. That is, they are also provided as a pure virtual base class and thus there must be subclasses for concrete implementation. CCs differ however in two main points: first, the supplied template parameters, i.e. their customization, is different and second, in the methods to be implemented. The template parameters of CCs target only types of input and output *command packages*. That is, one can specify the input and output type. CCs always only provide a single input and output port. The required methods in each implementation are *can_read*, *can_write*, *empty*, *pop* and *push*. *Can_read*, *can_write* and *empty* provide state information about the CC in conjunction with either a VC, the SCIF or the test running logic. *Pop* and *push* however are used for interacting with the CC. The functionality of these methods is explained with respect to the following example.

The naïve CC implementation is a FIFO queue connecting two VCs as depicted in Figure 2. In this configuration, the CCs input and output types are the same. In this implementation, the state functions *can_read* and *can_write* always return *true* whereas *empty* returns *true* or *false* according to the amount of elements kept in the queue. *Push* and *pop* would likewise move elements in and out.

Another example for a CC would be a queue introducing a probabilistic delay on the input. In general, this would work the same way as the simple queue CC. However, the *can_write* method would be modified such that it returns *true* or *false* with respect to some probability distribution.

Further examples of CCs are randomizing and size-limited queues as well as type converters.

4) Test run-logic

First, the test run-logic is responsible for holding all VCs and CCs in a centralized place. This is implemented as a set of C++ shared pointers. These so called smart pointers have a reference count by which it is ensured that the object does not get destroyed until there are no more usages. This also enables no explicit need for member definitions of VCs and CCs within the code. This significantly reduces the amount of code to be written.

The calling of the *process* methods of all VCs is another main objective of the run-logic. This is implemented by iterating over the complete set of all VCs and calling their *process* method once. Due to the nature of the implementation, the call will be redirected to the specific implementation of the VC. We also refer to this as one-step processing. That is, a VC does no more than a single step at a time. This ensures synchronicity with the clock which is fed to the DUT. Implicitly with the processing of VCs, CCs are also processed.

In order to control simulation progress, the run-logic ensures that i) the simulation is completed once it was truly successful and ii) the simulation does not run infinitely in case of an erroneous DUT (e.g. lost commands). To check whether the simulation is complete, the main processing loop OR's the *done* methods of all VCs and NOR's the *empty* methods of all CCs. If the result is *true*, the simulation is finished and will be stopped. For timeout control, a configurable timeout count is used which is decremented on every processing step. The timeout count can be reset from VCs if they do a meaningful action, e.g. if they receive a response or generate a *command packet*. Once the timeout count reaches zero, the simulation is aborted.

B. Detailed Description of the SystemC Interface

In contrast to the TF, the SCIF relies on *SystemC* data types instead of *command packages*. The reason for that is twofold.

First, the SCIF has to be connected to the DUT. Here, it is required to provide the exact same data types otherwise compilation would fail. We configured *Verilator* to use *SystemC* data types in order to provide a standardized interface. By this, the test harness can be exchanged to e.g. a full-fledged *SystemC* environment.

This can be useful to provide for instance system-level models based on *SystemC*. Within our implementation we chose to inherit from *sc_module* thus forming a *SystemC* module.

Second, the interaction between SCIF and DUT has to take place with respect to a clock signal. This is due to the fact the DUT does not provide a transaction-level interface but instead relies on the interface protocol being actually used. Therefore, the SCIF has to follow this protocol as well.

We will now describe the implementation of the SCIF. Due to the requirement of replicating the interface protocol while keeping up generality of the implementation, we chose to implement the SCIF with a state machine which is depicted in Figure 3.

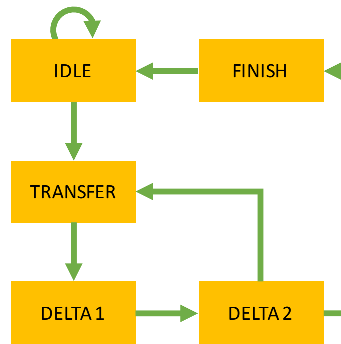

Figure 3. SCIF State Machine.

The state machine is partially synchronized with the clock from which the DUT is driven. That is, between certain states it is ensured that the state machine waits for a complete clock cycle until changing the state. This happens for instance for the transition between *TRANSFER* to *TRANSFER* via the *DELTA1* and *DELTA2* states or from *TRANSFER* to *IDLE* via *FINISH*. In all other cases the states are switched immediately to ensure that there are no extra cycles induced. Thus, if the receiver is always able to receive, the SCIF is able to drive a *data packet* each clock cycle to the DUT.

The initial state of the state machine is *IDLE*. It will stay in this state until there is a *data packet* to be transferred either from the SCIF to the DUT or vice versa. On this occasion the state machine switches to the *TRANSFER* state.

In this state the state machine obtains the *data packet* to be sent from the attached CC. The conversion of the original *command packet* to the corresponding *data packet* is performed by the CC, such that there is no additional information needed in the SCIF. In this state, it is up to the implementation how the bus protocol must be driven. The following state is named *DELTA1*.

Within the *DELTA1 and DELTA2* states it is ensured that the *SystemC* signals are applied correctly. If necessary, both states can be jumped by providing an alternate implementation. Usually, these states are only ensuring that a delta cycle is introduced into the running simulation. However, depending on the bus protocol the state following *DELTA2* can be either again *TRANSFER* or *FINISH*.

When the transfer was not successful, because e.g. the interface protocol was signaling that the receiver is not yet ready, the state machine will switch to *TRANSFER* again, keeping the *current data packet* applied to the interface. On the other hand, if the transfer was successful, the state machine switches to *FINISH*, cleaning up the internal state and eventually returning to *IDLE*.

IV. Example Test

This section describes how the introduced components are connected in order to form a concrete test. The DUT being tested provides a single request and response channel. For this description, it is not important to describe what the DUT exactly does. Instead, we are focusing on how the test applies data to the DUT and how it verifies its responses. However, it is important to note that the DUT is capable of sending *retry* commands on which the test has to react respectively.

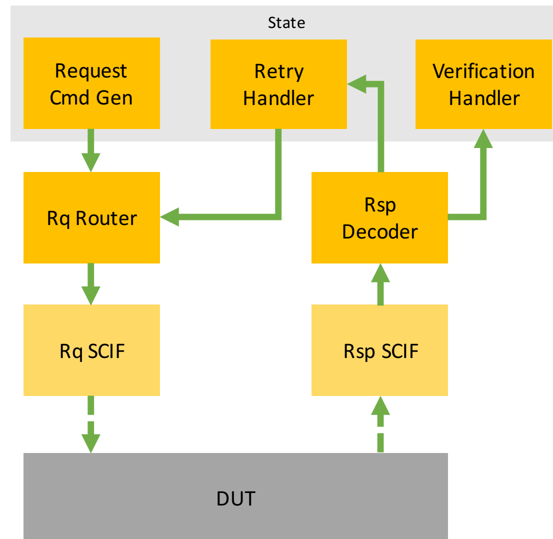

Figure 4. Block diagram of example test environment for a DUT consisting of a request and response channel with the capability of sending retry commands.

The overall test environment block diagram is depicted in Figure 4 showing that only few components are needed to build the test. We will now describe the test environment in detail.

A. Sending a Request

Sending requests requires generating *command packets*, which is done by the *request command generator*. This VC also updates the *state* of the simulation. That is, it will register the generated *command packet* with e.g. some meta data such that an incoming response can find its respective request command.

Next, the *request command packet* is transported via a CC to a 2:1 *request router*. This VC provides two input ports in order to handle *command packets* which need to be retried. Its routing algorithm therefore passes any incoming *command packet* on any input port to the singular output port.

Finally, the *command packet* is passed to the corresponding SCIF connecting the request port of the DUT to the test environment. That is, the SCIF receives the *command packet*, converts it to its low-level representation and forwards it with regards to the implemented bus protocol.

B. Response and Retry Handling

The handling of responses from the DUT to the test environment works in the reverse direction as the sending of requests does. First, the SCIF responsible for receiving responses from the DUT listens on the connected port. When the DUT signals the availability of data, the SCIF will apply the handshake signals necessary to receive the data. Subsequently, it converts the raw data to a corresponding *command packet* and moves this to the output port of the SCIF.

Next, a *response decoder* VC receives the *response command packet* and decodes it. The *response decoder* can be seen as 1:2 *router* moving the *response command packet* to the correct output port. Within the context of

this example, the decoding would result in making a decision whether the received response is a retry or not. Thus, it moves a non-retry response to the output where the *verification handler* is connected to. We describe the *verification handler* in the next section.

Likewise, if the response is a retry, it moves it to the output where the *retry handler* is connected to. The *retry handler* VC in turn has access to the state of the test and updates it. This update could be for instance the creating of a new *request command packet* with the exact same information as the original one but with new meta data assigned. This request is then passed again to the *request router* from which on it will be forwarded to the DUT.

C. Verification of Responses

The *verification handler* VC compares the DUT response to a reference model providing an expected response for a request. In conjunction with the test state, it becomes possible to implement complex reference models. For example, when verifying caches, the test would keep a history of requests and responses. By this, the reference model resembles the internal state of the DUT which makes it easier to verify incoming responses.

For example one can suppose the cache supports *read exclusive* commands. When the test issues multiple of these commands to the same address, then the data is not supposed to be changed because the cache owns the specific datum exclusively. By keeping a history of all responses, the *verification handler* can now monitor data changes. If a response with changed data is sent, this change would be detected and the test would abort due to data inconsistency. From this point on, a designer can cross-check with e.g. generated waveforms.

D. Introducing Delays in the Test Environment

If the test designer would use e.g. the naïve implementation of CCs, the test would be capable of sending requests without interruption. That is, the only delay on the request chain is given by the DUT when it is unable to accept more requests and thus forces the request SCIF to wait. However, this also causes the response chain to be always able to receive responses.

While the behavior for sending requests might be acceptable, it is usually not for responses because this would resemble an ideal scenario and might hide potential bugs. Thus, it is necessary to introduce delays on the response-chain. With the introduced TF this task is easily solvable, since the CCs provide e.g. the *can_write* method. Therefore, one has to replace the naïvely implemented CC between response SCIF and *response decoder* with one that does not always return *true* when *can_write* is checked. The approach of easily changing CCs to introduce e.g. delays provides the test designer more flexibility.

V. Results

A. Quality of the Verification Method

In order to prove that our verification approach works, we took measurements with respect to the line coverage and signal toggle coverage metrics. Specifically, *Verilator* generates this data by counting how often signals to the DUT were flipped and by annotating code lines in RTL source code which triggered these signal changes. Relating the amount of annotated code lines with the amount of total lines of code excluding comments leads to the *measured function coverage* metric.

Since the resulting percentage also contains signals flipped rarely and never triggered signals, we filter these out and obtain a cleaned coverage metric. Those signals are e.g. resets and specific address ranges which are not tested if not necessary. The identification and exclusion of those signals is done only once. Therefore, on subsequent simulation runs with enabled coverage measurement, the resulting metric is already cleaned.

Table I. Selected Coverage Metrics.

Test	Lines of Code	Measured Coverage
Module A	4655	92.0%
Module B	3668	92.0%
System A	31250	88.6%

Table I shows our measured coverage for two important modules and for a combination of modules. These are the most recent values.

During the design of our product, we first implemented coarse-grained tests which resulted in lower coverage. We then refined these tests to provide better randomized stimuli and more detailed verification checks. By this, we were able to increase coverage to the shown values. Note, that the tests currently do not apply illegal stimuli to the modules or test error recovery. Due to this, the measured coverage is below 100%.

B. Simulation Performance

To evaluate the performance of *Verilator* simulation, we re-implemented an existing test with SystemVerilog and simulated this with a selection of commercially available simulators. In this test, *Verilator* closely follows the performance of the commercial simulators when simulation runs with optimal settings, e.g. compiler set to high optimization level.

Furthermore, since a self-executable binary without license restrictions is created when using *Verilator*, we are able to run many simulations in parallel. This is technically possible with a commercial simulator but requires acquiring multiple licenses. As result, we are able to apply more parallelism in simulation with *Verilator* than with other commercial products. This evens out the performance penalty of the approach, especially with affordable many-core server CPUs. Using this strategy, we increased the probability to find critical bugs in our design. Running multiple simulations in parallel is also fully automated.

VI. Conclusions

As main conclusion presented in this paper we note that verification with C++ and Open Source Software works for highly complex hardware designs. The presented case covered a full-custom design of a data processing PCIe card.

By measuring line coverage and toggle coverage, we have shown that the quality of this verification method is high enough to rely on this verification method for complex FPGA-based designs. This metric can even be improved by tweaking test parameters and introducing more specific tests.

Eventually we note that the verification with C++ is a viable alternative to verification with, for example, SystemVerilog. In our opinion, this especially holds for startups and younger companies with certain budgetary restrictions. Furthermore, being able to run tens or even hundreds of tests simultaneously without license restrictions is an inherent advantage of the presented method. The presented approach enables reusability of modules which reduces development time for tests. Additionally, engineers with sufficient background in both hardware and software are capable of implementing tests with C++. This enables realization of correct verification from the very beginning. With the approach outlined in this paper, effective verification can be implemented even on a heavily constrained budget.

[1] W. Snyder, "Verilator and systemperl," in North American SystemC Users' Group, Design Automation Conference, 2004.

[2] W. Snyder, "Verilator Website," 2010. [Online]. Available: http://www.veripool.org/projects/verilator.

[3] R. Damaševicius and V. Štuikys, "High-Level Design of Soft IPs using C++ and SystemC," 2002.

[4] S. Park and S.-I. Chae, "A C/C++-based functional verification framework using the SystemC verification library," in 16th IEEE International Workshop on Rapid System Prototyping (RSP'05), 2005.