FEATURED IN THIS ISSUE:
The ISO 26262 Automotive Safety Standard focuses on verifying that a safety-critical design can continue to function correctly even in the presence of various faults. There are techniques that can be used to build and optimize the fault model to define the set of faults to be injected, with some impressive results.

See how the QVIP Configurator can be used to add Mentor Questa VIP components into processor verification environments. The Codasip automation flow can then generate an RTL and accompanying UVM verification environment from a high-level description of a RISC-V processor.

Go through the steps required to take advantage of Liberty libraries so that Questa PA-SIM can extract cell-level attributes that combine with UPF to model power-aware functionality.

In “Reset Verification in SoC Designs,” learn about several common reset tree verification issues and how to solve them.

Find out what to do when your formal verification run is unable to either prove an assertion exhaustively or find an error trace that violates the assertion.

Learn how PSS can be used to describe generic test intent for generating memory traffic in a typical SoC and how the separation built into PSS lets that single intent specification target different “implementation flavors” of the SoC.

When It Comes to Verification, Hitting the Wall Can Be a Good Thing.

By Tom Fitzpatrick, Editor and Verification Technologist

We recently upgraded my son’s phone. We were getting ready to upgrade anyway, but the thing that put us over the edge was that a couple of weeks ago his phone wouldn’t charge. Now, we had recently replaced the battery in the phone, so the fact that it wouldn’t charge was particularly concerning. We would have had to do the upgrade much sooner, but after a day of frustration, my son literally threw the phone against the wall and it started working. A visit to the tech center revealed that the charging problem was actually due to lint having collected in the charging cable connector slot, preventing the charger from connecting to the battery. The wall toss moved the lint just enough to complete the connection. The technician cleaned it out and, after upgrading my son’s phone, we gave his old one — with the brand-new battery — to my daughter. Of course, the process of backup-and-restore for both phones didn’t go as smoothly as the documentation would have suggested, but that’s another story. Sometimes you have to try something “off the wall” when the way you’ve been trying to solve problems doesn’t work. With that in mind, I am happy to share with you our Fall 2017 issue of Verification Horizons.

Our featured article for this issue is from our friends at IROC Technologies, who share their thoughts on “EDA Support for Functional Safety — How Static and Dynamic Failure Analysis Can Improve Productivity in the Assessment of Functional Safety.” As we’ve discussed before, the ISO 26262 Automotive Safety Standard focuses on verifying that a safety-critical design can continue to function correctly even in the presence of various faults. This article shows techniques that can be used to build and optimize the fault model to define the set of faults to be injected, and it shares some impressive results that you’ll be able to see in your environment.

“Sometimes you have to try something “off the wall” when the way you’ve been trying to solve problems doesn’t work.” —Tom Fitzpatrick

Next, our friends from Codasip, Ltd. present a “Step-by-step Tutorial for Connecting Questa® VIP (QVIP) into the Processor Verification Flow,” in which they show how QVIP Configurator can be used to add Mentor Questa VIP components into processor verification environments. Their automation flow can generate an RTL and accompanying UVM verification environment from a high-level description of a RISC-V processor. By making it easier to set up the QVIP component to fit into
the customized environment, the Configurator lets you take advantage of the built-in verification capabilities of the QVIP to strengthen the auto-generated UVM environment.

In “PA GLS: The Power Aware Gate-level Simulation,” my Mentor colleague Progyna Khondkar takes us through the steps required to take advantage of Liberty libraries so that Questa® PA-SIM can extract cell-level attributes that combine with UPF to model power aware functionality.

We next introduce a pair of formal-verification-related articles from members of our Mentor Formal Verification team. The first is “Reset Verification in SoC Designs,” where you’ll learn about several common reset tree verification issues and how to solve them. The authors also share some of the results that Questa’s Reset Check formal “app” can provide in addressing these issues.

In “Debugging Inconclusive Assertions and a Case Study,” we learn what to do when your formal verification run is unable to either prove an assertion exhaustively or find an error trace that violates the assertion. This often happens because the complexity of the design outpaces the formal tool’s capability, so you’ll see some tips and tricks to figure out why the analysis was inconclusive and reduce the complexity of the formal analysis so that you can fully verify the assertion.

Last but not least, my friend and colleague Matthew Ballance continues his series of articles discussing the new Accellera Portable Stimulus Standard (PSS) with “Getting Generic with Test Intent: Separating Test Intent from Design Details with Portable Stimulus.” In this article you’ll learn how PSS can be used to describe generic test intent for generating memory traffic in a typical SoC and how the separation built into PSS lets that single intent specification target different “implementation flavors” of the SoC. You’ll also see how you can specify the test intent independent of the actual design details, dramatically boosting your verification productivity.

My son is now back at college with his new phone. Everything seems to be working fine. Our only problem now is that my daughter’s playlists appear to be missing. I’m off to do a little more debugging. In the meantime, I hope you enjoy this issue of Verification Horizons.

Respectfully submitted,
Tom Fitzpatrick
Editor, Verification Horizons
CONTENTS

by Dan Alexandrescu, Adrian Evans and Maximilien Glorieux—IROC Technologies

Page 10: Step-by-step Tutorial for Connecting Questa® VIP into the Processor Verification Flow
by Marcela Zachariasova, Tomas Vanak and Lubos Moravec—Codasip Ltd.

Page 17: PA GLS: The Power Aware Gate-level Simulation
by Progyna Khondkar—Mentor, A Siemens Business

Page 20: Reset Verification in SoC Designs
by Chris Kwok, Priya Viswanathan, and Kurt Takara—Mentor, A Siemens Business

Page 26: Debugging Inconclusive Assertions and a Case Study
by Jin Hou—Mentor, A Siemens Business

Page 31: Getting Generic with Test Intent: Separating Test Intent from Design Details with Portable Stimulus
by Matthew Ballance—Mentor, A Siemens Business
EDA Support for Functional Safety – How Static and Dynamic Failure Analysis Can Improve Productivity in the Assessment of Functional Safety

by Dan Alexandrescu, Adrian Evans and Maximilien Glorieux—IROC Technologies

Integrated circuits used in high-reliability applications must demonstrate low failure rates and high levels of fault detection coverage. Safety Integrity Level (SIL) metrics indicated by the general IEC 61508 standard and the derived Automotive Safety Integrity Level (ASIL) specified by the ISO 26262 standard specify specific failure (FIT) rates and fault coverage metrics (e.g., SPFM and LFM) that must be met. To demonstrate that an integrated circuit meets these expectations requires a combination of expert design analysis combined with fault injection (FI) simulations. During FI simulations, specific hardware faults (e.g., transients, stuck-at) are injected in specific nodes of the circuits (e.g., flip flops or logic gates).

Designing an effective fault-injection platform is challenging, especially designing a platform that can be reused effectively across designs. We propose an architecture for a complete FI platform, easily integrated into a general-purpose design verification environment (DVE) that is implemented using UVM. The proposed fault simulation methodology is augmented using static analysis techniques based on fault propagation probability assessment and clustering approaches accelerating the fault simulation campaigns. The overall framework aims to: identify safety-threatening device features, provide objective failure metric, and support design improvement efforts.

We present a working example, implemented using Mentor’s Questa® Verification Platform where a 32-bit RISC V CPU has been subjected to an extensive static and dynamic failure analysis process, as a part of a standard-mandated functional safety assessment.

INTRODUCTION

Functional safety is a major enabler of many of the current and upcoming automotive products and has been formalized by the ISO 26262 standard. This body of work recommends methods and techniques to demonstrate that specific targets for failure (FIT) rates and fault coverage metrics (e.g., SPFM and LFM) are successfully met. In addition to expert design analysis, Fault Injection (FI) techniques can support design validation and verification efforts by providing trustworthy and actionable data while spending the least amount of computational resources and engineering effort.

On the hardware and system side, the ISO 26262 standard mentions or recommends fault injection in several locations in Chapter 4 - System Level, tables 3, 5, 8, 10, 13, 15, and 18 as a way to verify the correct implementation of the functional safety requirements and the effectiveness of a safety mechanism’s failure coverage at the system (i.e., vehicle) level. Chapter 3 – Hardware Level, table 11 also presents fault injection as a method to verify the completeness and correctness of the safety mechanisms implementation with respect to the hardware safety requirements. Chapter 3 further motivates the need for fault injection as a way to evaluate the efficiency of safety mechanisms for both checking the results against the SPF and the LF metrics and providing actual data for the computation of the residual FIT. Lastly, the Chapter 10 - Guideline on ISO 26262 presents fault injection (in addition to other methods) as an approach for the verification of the amount of safe faults and especially of the failure mode coverage.

Accordingly, the standard provides relevant guidance for a successful implementation and execution of fault injection campaigns. Firstly, the fault injection shall aim to prove that the safety mechanisms are working in the system (and not only in isolation). This requirement will obviously drive the need for a fault injection campaign performed on a representative
(i.e., close to reality) design abstraction. Secondly, the fault injection should exercise the safety mechanisms and extract data about their coverage. Lastly, the fault injection should highlight the behavior of the system in case of failures in order to provide the data required for the failure rate analysis and classification.

On a more practical note, fault injection campaigns are a significant undertaking, requiring engineering effort and computational resources. This process is costly and requires specific expertise, knowledge, and EDA support. This article presents a series of practical techniques that can be implemented in existing design and verification environments (DVE) using Mentor’s Questa® platform tools and aim at improving the confidence in the fault injection campaign results, minimizing the resources (CPU time, engineering effort) spent during the evaluation and maximizing the usefulness of the simulation results. The presented techniques are a part of the feature set of IROC’s comprehensive EDA platform for failure analysis of SoCs (SoCFIT).

**FAULT UNIVERSE BUILD-UP**

Fault simulation techniques are widely researched and used in electronic design practice. One of the most important challenges consists in dealing with the complexity of the system, the application, and the consequences of the fault.

Complex automotive systems have related complex fault universes. The ISO 26262 standard mentions random hardware faults; such as, caused by internal (technological manufacturing process, aging, alpha contamination) and external causes (electronic interference, single events, extreme operating processes). The corresponding fault models can be more or less precise and can be used only at specific design abstraction levels. Thus, a first challenge resides in the elaboration of a complete ISO 26262 fault universe for the considered circuit. While modern EDA could support a very low fault granularity (even at the transistor level) the cost of such an approach would be prohibitive. High-Level Synthesis (HLS) or RTL design representations are more appropriate and also correspond to the design level where most of the engineering effort is spent and where the required tools and environments – such as the DVE – are available. Therefore, the works presented in this article use the RTL description of the device under test (DUT) and the appropriate Questa® simulation tool.

The main challenge we will have to address consists in the relevancy of the HLS/RTL description with regard to the faults applicable to physical design features. In some cases, fault models can be attached to RTL elements, while in other cases, the RTL description doesn’t have a representation of the underlying hardware gates. As an example, a single event upset affecting a hardware flip-flop can be modeled in RTL as a fault attached to a Verilog register. However, a list of “single event transient” faults cannot be built as the RTL files have no indication yet of the logic network implementation.

Obviously, a gate-level netlist (GLN) description would have been easier to use, as the GLN instances correspond directly to design features. However, it is not always possible to perform fault simulation in GLN, as the verification designers may not have added gate-level simulation capabilities to the DVE. If such capabilities exist, it is overwhelmingly probable that the simulation speed is greatly reduced when compared to RTL simulations, leading to long (and costly) fault simulation campaigns.

We have built a tool to extract critical design features from RTL and GLN circuits: hierarchical information, list of signals and variables, list of circuit instances, and so on. The RTL and GLN databases are cross-linked. Based on a series of rules, RTL design elements are linked to actual GLN features whenever possible. The tool relies on VPI capabilities in addition to Questa® features such as the “find … -recursive” command. Any compound elements such as buses, arrays, and complex signal types are itemized to single physical signals whenever possible.

Moreover, each type of fault requires the corresponding fault rate that indicates how often the circuit encounters the fault in a given working environment. While the fault rate will be mostly used during the FMEDA efforts, it could be useful as a criterion for failure injection prioritization.
**FAULT UNIVERSE OPTIMIZATION**

We use a series of fault equivalence techniques to compact the fault universe.

State elements (sequential cells, memory blocks) should exist in both RTL and GLN. The tool is able to build an exhaustive fault list associated to state elements. However, combinatorial gates are not available in RTL. In this case, the tool uses a static analysis approach (based on the calculation of a fault propagation probability logic de-rating, or LDR, factor) that links the cell instances from combinatorial networks to the downstream state-element. According to the state of the circuit (the values of the signals and cell outputs), the propagation of the fault is subject to logic blocking. As an example, an AND gate with a “low” input will block any faults on the other inputs. In the context of this document, logic de-rating only covers the initial clock cycle propagation. The SoCFIT tool is able to compute LDR factors for very complex circuits (millions of flip-flops) in a reasonable amount of time (minutes).

LDR is able to link several circuit elements to a representative. Fault simulation can then proceed on the representative element and data for the equivalenced elements can be extrapolated back with the use of LDR factors.

Another technique for reducing the fault universe is clustering. RTL/GLN designs are explored in order to find elements that can be grouped together according to their likeness. The criteria can be structural (buses, hierarchical modules, naming conventions, distance) or functional, such as the probability for a fault in an element to propagate into another (LDR). Simulating only one representative of the cluster and generalizing the results to the full cluster will significantly reduce simulation times.

**SIMULATION-TIME OPTIMIZATION**

A vast majority of faults injected in the circuit will have no impact on the function of the system (safe faults). In this case, Check-point techniques can be very effective to identify simulation cases where the injected fault disappears quickly. We have developed a compact simulation trace approach based on CRC. First, we calculate the CRC for each flip-flop and memory in the design, each N clock cycle, during a golden, fault-free simulation (reference CRC trace file).

Then, we use the simulator capabilities to check the circuit status during the fault injection, and we stop the simulation as soon as the CRC matches the golden.

Hardware faults that can occur randomly, such as single event effects, can be also masked by temporal de-rating (TDR). TDR represents two different phenomena related to temporal aspects. The first consists in evaluating the effects of the fault with respect to the occurrence time within the clock cycle or application phase. The second aspect concerns only transient faults, such as SETs and copes, with the relevance of the pulse duration versus the clock period. TDR can be evaluated using low-cost probabilistic methods.

Memory blocks can contain data that is changed frequently. Soft errors affecting data stored in a memory block can only propagate if the affected data is actually read. If a write operation stores another data at the affected location, then the fault is overwritten. Accordingly, using the results of a single, golden simulation, we can compute a memory de-rating (MDR) factor which represents the probability for a fault to arrive during a sensitive opportunity window:

$$MDF = \frac{(t_{2} - t_{1}) + (t_{4} - t_{3})}{t_{S} - t_{1}}$$

![Figure 1](image-url)
We have built a manager tool that is able to select faults from the fault universe, prepare a simulation case including a fault injection feature, and run the case using the Questa® simulation tool. The actual fault injection can be performed through simulator capabilities such as force/release or PLI/VPI procedures. The simulation case will be run and the existing DVE features will highlight the outcome of the fault injection, providing the data required for further failure analysis.

The article “Extending UVM Verification Models for the Analysis of Fault Injection Simulations”, published in the Verification Horizons edition from June 2016, shows how the components of a UVM functional verification environment can easily be extended to record additional information about the types of errors that have occurred. This additional information can be used to classify failing tests based on their system level impact.

In the context of the ISO 26262 standard, the failures need to be classified in the following categories: “safe fault”, “detected multiple-point fault”, “perceived multiple-point fault”, “latent multiple-point fault”, or “residual fault/single-point fault.” It’s important to understand if a fault simulation campaign can fulfil this requirement.

After analyzing Figure B.2 from the ISO 26262 document, we can conclude that failure classification should be able to bin the observed faulty circuit behaviors to the ISO 26262 classes, but it will have to be complemented by design review and, specifically, by the identification of safety mechanisms.

As an example, “safe faults” correspond to the category of faults that are either a) not affecting a safety-related element or b) affecting a safety element but cannot lead to a safety goal violation in combination with an independent failure.

A fault simulation campaign can help with a), as it will provide the percentage of faults that caused no ill effects. The b) case requires fault simulations with multiple fault injections, which will be more costly.

“Single-point faults” have the potential to cause issues, and they are not covered by safety mechanisms. Fault simulation will be able to help with the evaluation of this class of failure, but it will have to be helped by design review.

In any case, the outcomes of the fault simulation campaign are twofold:

- Quantitative results, such as the percentage of failures in the different classes to be used in the FMEDA efforts
- Qualitative results about the behavior of the circuit in the presence of faults

**DEVICE UNDER TEST AND FAULT SIMULATION CAMPAIGN RESULTS**

We have prepared a DVE consisting of the PULPino RISC-V core, the related simulation environment, as well as runtime software consisting of a bit counter application. The DVE has been enhanced with modules of IROC’s SoCFIT reliability evaluation platform, running on top of Mentor’s Questa® simulation tool. The platform is able to build a complete fault universe; inject faults in the DUT; schedule, execute, and manage the fault simulation campaign; monitor circuit behavior in the presence of faults; and evaluate and finally classify failures. The failures observed during the simulation campaign are natively reported in terms of “classical” silent data corruption (SDC) and detected unrecoverable error (DUE). SDC failures correspond to cases where the workload executes to completion but the results are false. The DUE category is comprised of cases where the simulation didn’t complete successfully: the CPU hangs, loops, or generates exceptions.

In the default configuration, there are no safety mechanisms (in the ISO 26262 sense) available on PULPino, and we consider that the CPU is a safety-related HW element. Accordingly, the injected faults could be mapped to ISO 26262 single-point faults (when a SDC/DUE has been generated) and safe faults (when the simulation completed successfully) categories. If we assume that the CPU is integrated in an automotive system that features a safety mechanism able to detect major CPU failures (such as a watchdog circuit), then the DUE failures can be managed through a reset/reload. Accordingly, DUE
failures can be categorized as multi-point faults. SDC failures cannot be detected by the watchdog, as the CPU is running and able to service the watchdog requests without any issues. Accordingly, SDC failures can be categorized as single-point faults. In addition to failure classification, the fault injection campaign also allows the calculation of the percentage of safe faults (i.e., faults with no impact on the application).

The simulation campaign consisted in more than 22,000 simulation cases. Transient hardware faults of type single event upset (SEU) have been injected fairly in all the elements of the CPU. The results are presented below (95% confidence intervals are indicated inside the parenthesis):

<table>
<thead>
<tr>
<th>Observed Failures</th>
<th>Lower Limit</th>
<th>Upper Limit</th>
<th>- %</th>
<th>+ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.03</td>
<td>5.57</td>
<td>-97.5%</td>
<td>457.2%</td>
</tr>
<tr>
<td>2</td>
<td>0.24</td>
<td>5.57</td>
<td>-87.9%</td>
<td>178.6%</td>
</tr>
<tr>
<td>5</td>
<td>1.62</td>
<td>10.24</td>
<td>-67.5%</td>
<td>104.8%</td>
</tr>
<tr>
<td>10</td>
<td>4.80</td>
<td>17.08</td>
<td>-52.0%</td>
<td>70.8%</td>
</tr>
<tr>
<td>20</td>
<td>12.22</td>
<td>29.67</td>
<td>-38.9%</td>
<td>48.4%</td>
</tr>
<tr>
<td>50</td>
<td>37.11</td>
<td>64.78</td>
<td>-25.8%</td>
<td>29.6%</td>
</tr>
<tr>
<td>100</td>
<td>81.36</td>
<td>120.53</td>
<td>-18.6%</td>
<td>20.5%</td>
</tr>
<tr>
<td>200</td>
<td>173.24</td>
<td>228.65</td>
<td>-13.4%</td>
<td>14.3%</td>
</tr>
<tr>
<td>500</td>
<td>457.13</td>
<td>544.77</td>
<td>-8.6%</td>
<td>9.0%</td>
</tr>
<tr>
<td>1000</td>
<td>938.97</td>
<td>1062.92</td>
<td>-6.1%</td>
<td>6.3%</td>
</tr>
</tbody>
</table>

Another practical aspect that needs to be considered are the number of simulations required to obtain statistically-significant data. In the presented SEU fault simulation scenario, we have used a 95% confidence interval together with a Poisson statistical model for the event rate calculations. The choice of this model was justified by the fact that the Poisson distribution is perfectly fit for the observation of radiation-induced failures that are discrete events occurring in an interval of time with a known average rate, independently of the time since last event. A 95% confidence interval predicts that the true value of the tested parameter has 95% probability to be within the computed lower/upper limits. The following table provides practical examples of pre-compute limits and percentages:

<table>
<thead>
<tr>
<th>Total Fault Injections per Design Flip-Flop</th>
<th>Single Point Faults %</th>
<th>Multi-Point Faults %</th>
<th>Safe Faults %</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 Fl / FF</td>
<td>1.79% (1.62% ~ 1.96%)</td>
<td>12.92% (12.46% ~ 13.39%)</td>
<td>85.29% (84.10% ~ 86.50%)</td>
</tr>
</tbody>
</table>

The table indicates that for given random, fair, fault injections in the circuit, we need to observe at least 100 failures if ±20% bounds for a 95% confidence interval is required. In our example, the SPF category had 404 failures, which is sufficient for ±10% bounds.

Lastly, it may be of interest to understand what CPU elements are critical for a specific or for multiple applications in the case of a CPU-type device that may be used with different workloads in different products. To address this problem, the CPU can be exercised with several applications and the elements’ susceptibility (ability to cause failures) can be presented using the following format:

The workload-specific failure data can be then exploited to evaluate the list of design elements that are critical for all workloads and that can be prime targets for any hardening or protection efforts.
CONCLUSIONS

We have presented a fault injection environment (FIE) – built on Mentor’s Questa® simulation tools and IROC’s reliability evaluation platform – that is able to evaluate the fault susceptibility of complex microelectronic designs. The presented techniques can be implemented in existing DVEs and deliver results usable in the context of standard-driven reliability assessment and management efforts.
Verification Intellectual Properties (VIPs) play a very important role in the verification flow of modern SoCs. They can check the correctness of communication over system buses and provide master, slave, decoder, or arbiter components if these are missing in the verification set-up. This article describes verification of RISC-V processors, focusing on the combination of automatically generated UVM verification environments by QVIP Configurator and Questa® VIP (QVIP) components. The section with step-by-step instructions will demonstrate how to add QVIP components into processor verification environments.

**RISC-V**

RISC-V is a new processor architecture with a free to use, modern and open Instruction Set Architecture (ISA) definition. It was originally designed by the University of California, Berkeley, to support research and education. Eventually, RISC-V expanded and established itself in the industry, thanks to the RISC-V Foundation which is now covering most of the activities related to RISC-V.[1]

RISC-V standard includes the definition of base integer instruction set ("I" for 32 general-purpose registers or "E" for 16 general-purpose registers) and optional ISA extensions, such as multiplication and division extension ("M"), compressed instructions extension ("C"), atomic operations extension ("A"), floating-point extension ("F"), floating-point with double-precision extension ("D"), floating-point with quad-precision extension ("Q"), and other experimental and custom extensions. However, RISC-V does not include any hardware microarchitecture definitions, which means that IP vendors can implement and sell varied versions of RISC-V processor IPs with distinctive features and multiple bus interfaces.

This article describes verification of communication of the RISC-V core with its surrounding components using QVIP. RISC-V compliant processor cores and components referenced in this article were implemented by the Czech-based IP company Codasip.[2]

**AUTOMATICALLY GENERATED UVM VERIFICATION ENVIRONMENTS**

One of the main benefits of the Codasip solution is that apart from an off-the-shelf RISC-V IP series Codix Berkelium (Bk), the company also provides an EDA tool for processor development and customization. The tool describes processors at a higher abstraction level in Architecture Description Language called CodAL. From this description, many outputs are automatically generated, including the RTL representation of the processor, a complete set of software tools such as compilers, linkers, simulators, debuggers and profilers, and an UVM verification environment with a random assembler programs generator. This allows for very fast customization and optimization of the cores according to the user’s individual needs.

From a verification perspective, the RTL generated for a specific RISC-V core serves as the Design Under Test (DUT). The Codasip automation flow can also generate functional instruction-accurate reference models. “Functional” means that the reference description does not contain micro-architectural details like timing, pipelining, or any kind of hardware blocks. For illustration, see Fig. 1.
For each generated assembler program, DUT outputs are automatically compared to those of the reference model. Consequently, the generated UVM verification environment is processor-specific, and for the purposes of UVM generation, all necessary information is extracted from the high-level processor description in CodAL.

The generated verification environment has a standard UVM-style structure. For the purpose of this article, a detailed description is not needed, so only a simple block scheme is provided in Fig. 2. It illustrates the relation of the Berkelium core (DUT), connected main memory, and several UVM agents:

- ASIP agent – the main processor agent. It includes assertion checkers and coverage monitors for the processor ports and some internal signals. It also handles driving of input ports, mainly interrupt ports.
- Registers agent – monitors access to internal registers of the processor.
- Decoder agent – monitors coverage of executed instructions and combinations of instructions.
- Memory agent – handles loading of test programs to the program part of main memory, and monitors access to the memory while instructions are executed.

For further reference, it is important to note that the Berkelium core interface (for connecting the memory and optional peripherals, not depicted in Fig. 2) is an AMBA-like interface.

**QUESTA® VIP CONFIGURATOR**

For easier integration of QVIP, there is a tool called QVIP Configurator. The tool is provided in the installation package along with supported protocol VIPs, and it can be used to create QVIP test benches based on UVM, which include a basic set of test bench building blocks. The steps to generate the test-bench are simple: After opening QVIP Configurator, required testbench components can be added and configured, such as instances of DUTs, modules, VIP protocols, and memory modules in the testbench project. It is also possible to set the clock and reset modules, define address mapping, and select available sequences.
Step 1
First of all, the DUT component must be added. Let’s use a Codix Berkelium IP core in this test case. To add the DUT module, navigate to the menu “File > Import a DUT / Module”, or use the icon shown in Fig. 3.

Step 2
Next, VIP components for AHB-lite protocol are added. This can be done via the menu “Actions > VIP Instance”, or by right-clicking in the workspace and selecting “Create VIP Instance”. The Create window is shown in Fig. 4. Two newly added components (QVIP Monitor and UVM Agent) are generated in one step as the Monitor component is configured and driven by its UVM agent.

Step 3
Now we need to establish connection between the DUT and the QVIP monitor component. For this step, it is recommended to use the Connect Instance wizard with module selection for components that shall be connected. To launch the wizard, right-click the QVIP component in the workspace and select “Connect Instance”.

Step 4
When steps 1–3 are completed, the testbench is ready to be generated. To start, navigate to the
menu “Actions” and select “Generate Testbench”. An example of a fully connected test environment is shown in Fig. 6.

**QVIP AS MONITOR (PROTOCOL CHECKER)**

One of the main purposes of QVIP is to verify that communication over a system bus complies to the communication protocol. This task is performed by assertion checkers and protocol coverage ensured by comprehensive QVIP inbuilt sequences. There are versions of QVIP that support various industry standard interfaces, see the overview in Fig. 7.

A step-by-step tutorial follows on how to connect the QVIP protocol checker for AHB-Lite bus connection between the Berkelium core and the main memory. All steps are illustrated by the block scheme in Fig. 8 on the following page. The example is based on the testbench generated from QVIP Configurator as described in the previous chapter. The method can be used also for very complicated SoCs.
1. First, a path to Questa® VIP package must be set as an environment variable.

   ```bash
   export QUESTA_MVC_HOME=<PATH_TO_VIP_INSTALLATION>/Questa_VIP_10_6_b
   ```

2. In our example, we use Questa® as the main RTL simulator. When running Questa® with QVIP, parameter -mvchome must be used with the vsim command.

   ```bash
   vsim -gui -mvchome <PATH_TO_VIP_INSTALLATION>/Questa_VIP_10_6_b -do "do questa/start_gui.tcl -codix_berkelium xexes/crc.xexe"
   ```

3. Compilations of the QVIP package and the AHB package are part of a compilation script that needs to be created. In our example, the script is called `start_gui.tcl`. When another protocol is used, then another QVIP package is compiled.

4. As indicated by the generated testbench code, components participating in the connection of QVIP instantiate QVIP interface and QVIP monitor module. Then the QVIP monitor signals are connected with the DUT.

5. It is important to set the agent configuration in the UVM environment configuration object. Below, there is an example of the generated configuration `ahb_lite_monitor_0_cfg` integrated to our UVM environment.

   ```bash
   // ahb agent configuration
   ahb_lite_monitor_0_cfg = ahb_lite_monitor_0_cfg;
   
   // Constructor - creates new instance of this class
   function new( string name = "m_codix_berkelium_ca_env_config_h" );
   super.new( name );
   ahb_lite_monitor_0_cfg = new();
   endfunction new
   ```

6. In our UVM environment, an AHB QVIP agent must be created. The package is `mgc_abh_v2_0_pkg`.
7. Now we need to configure the AHB QVIP agent. All parameters can be set directly in the QVIP Configurator (see Fig. 9), and after testbench generation, they are located in the files `top_params_pkg.sv` and `<VIP_instance_name>_config_policy.svh`. The `is_active` parameter is set to 0 when just the protocol checker is in use. If QVIP supplies a master, a slave, a decoder or an arbiter, the parameter is set to 1. In such case, the `agent_type` parameter should be set either to AHB_MASTER, AHB_SLAVE, or AHB_ARBITER. The parameter `if_type` represents the type of the AHB interface, and it can be set to AHB, AHB_LITE, or ARM11_AHB. The parameter `en_cov` enables coverage collection, the parameter `en_logger` enables creation of a log with transactions, the parameter `en_sb` enables checking that the written data are read correctly, and the parameter `en_txn_ltnr` enables printing transactions into the simulation transcript.

**QVIP AS MASTER/SLAVE**

The previous chapter explained how to connect QVIP as a Monitor (Protocol Checker). There are more options to connect QVIP, for example as a Master or Slave component, if such components exist in the verified environment.

The scenario with QVIP as a Master component (shown in Fig. 10) is suitable for verification of existing Slave DUT components, for example AHB memory. The QVIP Master is equipped with test sequences, cover-points and assertions, like in the Monitor scenario. After running the testbench generated from QVIP Configurator, it is therefore possible to verify that the Slave DUT component is compliant with the tested communication protocol.

**UVM ENVIRONMENT**

![Figure 10: QVIP Connected as a Master Component](image)

In the second scenario, QVIP is connected as a Slave component. In the default mode, it will simulate the behavior of a memory - the behavior can be set in Agent configuration. Successful testing guarantees that the Master DUT’s communication protocol is compliant.

**UVM ENVIRONMENT**

![Figure 11: QVIP Connected as a Slave Component](image)
CONCLUSION
This article provided a step-by-step tutorial for connecting QVIP into the processor verification flow. The tutorial explained the use of QVIP Configurator and described the crucial parts of the generated test bench, essential for debugging when misbehavior is reported. The main benefits of using QVIP and its Configurator are as follows:

- Efficiency: Replaces implementing complex UVM agents for each new bus protocol.
- Automation and rapidity: QVIP Configurator speeds up assembly of the verification environment as it generates the testbench automatically.

Codasip connected AHB QVIP to their RISC-V compliant Berkelium processors and, similarly to QVIP Configurator, automatically initiated and connected the QVIP agents.

REFERENCES

PA GLS: The Power Aware Gate-level Simulation

by Progyna Khondkar—Mentor, A Siemens Business

In post-synthesis, gate-level netlist (GL-netlist), power aware (PA) simulation, the fundamental focus is to identify PA specific cells already present in the netlist. The associated UPF with the netlist design, determines the supply network and power connectivity to these special PA cells, and aid to keep their outputs from being corrupted. Hence, the GL-netlist-based power aware simulation (PA-SIM) input requirements are mostly the same as for RTL simulation. However, the design-under-verification at this stage is the GL-netlist from synthesis, so logic gates from standard, multi-voltage (MV), and Macro cell Liberty libraries are already inserted or instantiated in the design. Hence PA-SIM at post-synthesis also requires Liberty libraries as input in order to accumulate different cell-level attributes and power down functions. The Questa® PA-SIM tool utilizes the libraries to extract different attributes and functionalities which can be summarized as follows.

**List 1.1 Liberty Libraries Required at GL-Netlist PA-SIM**
- Identify a special power aware cell
- Conduct power aware simulation based on the cell identification along with UPF specification
- Apply appropriate corruption semantics accordingly

One significant aspect of GL-netlist PA-SIM is that all cell instances are interpreted as containing drivers because these cells are usually leaf level cells or they are an instance that has no descendants. As a result, buffer cell instances in a GL-netlist will cause corruption when powered down. This contrasts with RTL Verilog ‘buf’ primitives that do not represent drivers and, therefore, do not cause corruption when powered down, but they do not isolate and may propagate corrupted signals from upstream drivers.

During GL-netlist PA-SIM, corruption will occur on the output ports and sequential logic of any detected gate-level cells. In addition, power aware simulation automatically treats a module as a gate-level cell if the module contains the `celldefine attribute or the `specify blocks in HDL code. Even these cells are not defined in the Liberty syntax; the standard processing of driver-based corruption is still applied to these cells, similar to that for RTL cell designs.

UPF 1801-2013 or UPF 2.1 LRM provides a dominant mechanism to devise a driver-based corruption of any HDL cell—even when there is no `celldefine or `specify block—through the set_design_attribute [-attribute (name value)]* command. PA-SIM treats all module, entity, or design elements as a gate-level or leaf cell when the following syntax is applied on them.

**Example 1.1 Leaf-level or Gate-level Cell Treatment of Design for Driver-Based Corruption**

Define through UPF File:
set_design_attributes -models FIFO
-attribute (UPF_is_leaf_cell TRUE)

Define through HDL Annotation:
SystemVerilog or Verilog Attribute Specification:
(* UPF_is_leaf_cell="TRUE" *)
module FIFO (<port list>);

VHDL Attribute Specification:
attribute UPF_is_leaf_cell : STD.Standard.String;
attribute UPF_is_leaf_cell of FIFO : entity is “TRUE”;

Though the latest UPF 1801-2015 or UPF 3.0 LRM revised the syntax for leaf-level and gate-level cell
definition to enable driver-based corruption through `UPF_is_hard_macro` instead of `UPF_is_leaf_cell` attributes, the semantics and use model remain identical.

During GL-netlist PA-SIM, apart from detecting the standard and Macro cells and applying corruption accordingly, the simulator is also required to automatically identify special power management or MV cells—such as, isolation (ISO), level-shifter (LS), and retention flops (RFF)—in the design. The detection of MV cells is primarily done through the cell-level attributes available in the corresponding Liberty libraries and are usually cross-compared with the corresponding definition of strategies in UPF. Recalling the syntax and examples of ISO, LS, and RFF from UPF LRM, and also the fact that a GL-netlist contains at least ISO, LS, and RFF cells through synthesis, most of these cells are already specified either through following UPF commands and options or through tool auto detection processes.

**Example 1.2 ISO, LS, RFF Automatic Detection in GL-netlist Simulation through UPF Command**

ISO cells:

```
set_isolation strategy_name [-instance {{instance_name port_name}*]]
```

Where the `<instance_name>` is a technology leaf-cell instance and the `<port_name>` of the logic port that it isolates.

LS cells:

```
set_level_shifter strategy_name [-instance {{instance_name port_name}*]]
```

Similarly where the `<instance_name>` is a technology library leaf-cell instance and the `<port_name>` of the logic port that it level-shifts.

RFF cells:

```
set_retention retention_name [-instance {{instance_name [signal_name]}*]
```

Here in this case as well the `<instance_name>` is a technology library leaf-cell instance and the optional `<signal_name>` is the HDL signal that controls retention. If this instance has any unconnected supply ports or save and restore control ports, then these ports need to have identifying attributes in the cell model, and the ports shall be connected in accordance with this `set_retention` command.

In GL-netlist PA-SIM, the tool’s auto detection process of MV cells actually refers to the cells that are not specified through the `-instance` but through Liberty or other attributes. Hence, for the rest of the cells that are not specified in the UPF file, GL-netlist PA-SIM automatically detects the right UPF strategy to which they belong and treats them in a similar way to cells of that strategy specified with an `-instance` argument. Questa® PA-SIM detects power management cells based on one of the following.

**List 1.2 Liberty Cell-Level Attributes**
- `is_isolation_cell`
- `is_level_shifter`
- `retention_cell`

**List 1.3 Library Cell Name from UPF Commands**
- `map_isolation_cell` `isolation_name` [-lib_cells lib_cells_list]
- `map_level_shifter_cell` `level_shifter_strategy` [-lib_cells list]
- `map_retention_cell` `retention_name_list` [-lib_cells lib_cell_list]
- `use_interface_cell-cell-strategy` list_of_isolation_level_shifter_strategies [-lib_cells lib_cell_list]

Note that `map_isolation_cell` and `map_level_shifter_cell` are deprecated from UPF LRM 3.0 with `use_interface_cell` command. Unlike `map_isolation_cell` and `map_level_shifter_cell`, the `use_interface_cell` can be used to manually map any ISO, LS, or combined isolation level-shifter (ELS) cells.

**List 1.4 UPF name_format Command for Defining Names of Implicit Objects**
- `[isolation_prefix string]`
- `[isolation_suffix string]`
- `[level_shift_prefix string]`
- `[level_shift_suffix string]`

**List 1.5 Synthesis Pragmas**
- `isolation_upf`
- `retention_upf`
Although the GL-netlist PA-SIM does not have any exceptions from the fundamental concept of PA-SIM at the RTL, the Questa® PA-SIM tool procedure requires additional commands to process the information discussed above (Lists 1.2 and 1.3).

**Tool Procedures for Liberty Processing at GL-netlist:**  
Compile: No change  
Optimize: requires including either  
“vopt -pa_libertyfiles” or  
“vopt -pa_dumplibertydb”  
Simulate: No change

The following list explains the Liberty library referencing methods for GL-netlist PA-SIM.

**List 1.6 Liberty Referencing in PA-SIM at GL-netlist**

- **pa_libertyfiles-**  
  Specifies the Liberty files to read.  
  It is also possible to specify multiple files by separating file names with a comma.  
  e.g., vopt -pa_libertyfiles=a.lib,b.lib

- **pa_dumplibertydb-**  
  Specifies the name of the Liberty attribute library database for future reference.  
  e.g., vopt -pa_dumplibertydb=lib_datafile

Apart from detecting standard, Macro, and MV cells, PA-SIM also is required to virtually infer missing MV cells in the design. In general the virtual inferring process is limited to RTL where physical MV cells are not instantiated yet. Inferring may also be required at the Mixed-RTL, where some of the MV cells are still missing. Hence during GL-netlist, such virtual inferring is redundant. However, PA-SIM provides user controllability through tool procedures where it is possible to control the inference.

**Tool Procedures for Controlling Virtual Inerring of MV Cells:**  
Compile: No change  
Optimize: -vopt requires adding one of the following to disable auto inference:  
“vopt -pa_disable=insertls” -  
Disable LS cell insertion  
“vopt -pa_disable=insertret” -  
Disable RFF cell insertion  
Simulate: No change

Using one of the above procedures, based on the requirements, will allow the tool not to infer the appropriate cells virtually at any design abstraction level. But since physical MV cells are already inserted in post-synthesis GL-netlist designs, hence using a tool procedure during optimization as follows will instruct the tool to disable all three virtual insertions for ISO, LS, and RFF at once.

**Required Tool Procedure for GL-netlist PA-SIM:**

Optimize: vopt- requires to include “vopt -pa_gls”

Hence the GL-netlist PA-SIM as well as Mixed-RTL mechanisms can be summarized as follows:

**List 1.7 Summarization of PA-SIM Mechanism for GL-netlist and Mixed-RTL Design**

- Detect standard and Macro cells in the netlist and perform corruption based on the driver, UPF strategies, or power down functionalities from Liberty.
- Detect MV cells in the netlist and match them with corresponding UPF strategies at that point in the design.
- Virtually infer MV cells, if missing in the netlist, based on UPF strategies.
- Conduct automated power aware sequence checks and testbench-based simulation similar to RTL PA-SIM.

So once the cell detection or inferring process is completed as discussed above, the tool conducts power aware simulation on the GL-netlist similar to the RTL design. Although the Liberty file is required as additional input in the GL-netlist, however it is recommended to use the same testbench from RTL stage to confirm verification consistency.
Modern system-on-chip (SoC) designs contain a high level of complexity in the reset distribution and synchronization circuitry. Verifying that a design can be correctly reset under all modes of operation presents a significant challenge. In this article, we present the commonly occurring issues that are involved in reset tree verification and solutions to address them.

INTRODUCTION

Today’s SoC designs integrate many design IP blocks from various providers, each with their own implementation of reset. The reset architecture of a digital design can also be very complex. Designs have multiple sources of reset, such as power-on reset, hardware resets, debug resets, software resets, and watchdog timer reset. Errors in reset design can lead to metastability, glitches or other functional failures of the system. Furthermore, complex interactions can occur with the convergence of multiple resets, multiple clocks and multiple power domains. In many cases, this leads to a reset tree that is larger and more complex than the clock tree. Many of the concerns related to clock tree synthesis and load balancing now apply to the reset tree as well. Clearly, it’s a challenge to ensure that all sources of reset propagate safely to the intended destinations under all conditions!

Traditionally, simulation has been the primary method used to verify reset behavior, often with a heavy reliance on gate level simulation. However, RTL-level simulation testing is often incomplete, and gate level simulation can only be run late in the design cycle. Even worse, typically reset-related bugs are of a very serious nature, rendering the chip completely unusable. For example, a reset bug may prevent the reset of the design to a known good starting state, making its operation completely unreliable. In more extreme cases, the design may consume too much power during assertion of reset, causing the device to overheat and be permanently damaged. All of these factors negatively combine to cause costly, late-stage design changes; and, in the worst case, multi-million dollar silicon re-spins and time-to-market delays.

COMMON PROBLEMS

We first highlight some of the common reset architecture issues that we have seen in our experience on real world designs. We will separate it into two main categories: (a) issues related to correctness of reset trees and (b) issues related to the usage of resets.

A. Reset Distribution Tree

The first set of reset design issues are related to the incorrect implementation of resets. These problems are usually detected in the reset tree. Figure 1 shows an example—if this reset tree is incorrect, then all the other checking based on the reset sourced in the design will be incorrect, and the chip will not function properly. There is a long list of potential problems that we have seen, and we highlight two common ones here.

In general, resets can only be defined as asynchronous or synchronous to a clock. Sometimes, it’s being used as synchronous reset for clock1, and asynchronous for clock2. This usually indicates a misunderstanding of the reset in the system. Figure 2a shows a simple schematic in which a reset is used as both asynchronous and synchronous.

Sometimes while the design is evolving, wrong logic could be inserted into the reset tree. Incorrect logic can come in different forms. Some common problems include...
include the addition of tristate buffers and gates such as XOR gate, as shown in Figure 2b and 2c.

To describe this set of problems, we need to first define the terms “reset domain” and “clock-domain”. A reset domain is characterized by the following attributes – a) Type - synchronous or asynchronous b) Polarity - active low or active high c) Value - set for 1 and reset for 0 and d) the top resetting signal. Multiple synchronous reset domains can be grouped together. Similarly, a clock-domain indicates the clock source of a given register and optionally, the clock polarity. Multiple synchronous clock sources can be grouped together in a single clock-domain.

In a design with multiple reset and clock-domains, it is important to verify that resets are used properly in the context of the clocking scheme design. Specifically, the reset tree annotates the sequential elements in the design with the reset domain information. The clock tree is generated and the clock-domain information is annotated on every register. With every register in the design having a clock-domain and reset domain, the usage of the resets in a particular clock-domain can be determined. With this information, the reset analysis can identify asynchronous reset crossings between sequential elements in the same clock-domain, as well as identify crossings between asynchronous and synchronous resets between registers in the same clock-domain. These paths between resets are called Reset Domain Crossing (RDC) paths.

All asynchronous reset signals should be synchronized to the target clock-domains before being used. If the reset is de-asserted (released) asynchronously, it may violate the reset recovery
time leading to metastability. When an asynchronous reset violates the setup and hold requirements for a register, the metastability due to reset will result in a random delayed release of reset on the metastable registers. Figure 3 shows the waveform for the asynchronous reset problem.

To protect the circuit from metastability due to reset, reset synchronizers needs to be inserted. Figure 4a and Figure 4b show 2 typical reset synchronizers. The external reset signal, RST_n, is active low. When it is asserted, the output of the reset synchronizer will be ‘0’ immediately. When RST_n is released, the output of the reset synchronizer will be ‘1’ in the next or the cycle after. The release of the reset signal is synchronized to the clock, CLK. The synchronizer prevents timing violation when the reset is deasserted. Figure 4c shows the waveform at the output when synchronizer is inserted. Note that there is no metastability, and the deassertion at Q is delayed by one cycle.

Once resets are synchronized, designers must ensure that the reset signals are used properly in downstream logic. Two common problems are that resets are used with wrong polarity or clock. Figure 5a shows that the downstream register using the synchronized reset is clocked in a different clock (CLK2) from reset synchronizer (CLK). Clock-Domain Crossing (CDC) analysis will detect a CDC crossing and the problem will be identified during CDC verification. Figure 5b shows that the register reset by the synchronizer is using active high reset, while the synchronizer is an active-low reset. Since this is not a CDC crossing, this error will not be caught by CDC tools, and cannot be caught easily.

C. Reset Domain Crossing (RDC)
The third set of reset problems is related to reset domain crossing (RDC) paths. Most modern chips employ multiple clock-domains, and metastability in such designs due to these asynchronous clock domains “crossing” is a known issue. Advanced tools are available to catch such structural and functional CDC issues. However, even if the data
path is in the same clock-domain, if the reset of the source flop is different from that of the destination register, this asynchronous crossing path can lead to metastability at the destination register. This crossing involves understanding the use of reset domains and their interaction with clock-domains.[6][7]

The simplest case of RDC is when transmit and receive flops were reset by signals belonging to different reset domains. Figure 6a shows a simple RDC path. If RST1 is asserted while RST2 is not asserted, DFF2 can sample asynchronous data.

If the data transition is within the setup and hold window, then the DFF2 output will become metastable, as shown in Figure 6b.

Design structures used in reset architecture can mitigate RDC problems. One of the RDC synchronization methods is to isolate the input of the receiving domain from the output of the source domain. This requires enable signal to be generated in receiving clock and reset domain which isolates the input of the receiving register when the asynchronous reset of the transmitting domain is asserted. Isolation by gating can happen through the data path or the clock path of the receiving domain, as shown in Figure 7.
In general, chips partition resets through the use of delays in phases to avoid power surges which could burn the chip. This could lead to another problem. Figure 8 shows a simple example. The assertion of RST will cause DFF2 to reset while DFF2 is still in functional state. The inconsistent reset delay between adjacent registers causes incorrect data at the downstream logic. The reset structure assumes that RST is asserted for more than 3 cycles. If RST is only asserted 2 cycles, then when RST deasserts, DFF2 will be corrupted by DFF1 before DFF1 gets the reset signal.

VERIFICATION RESULTS

Because the metastability induced by the reset circuitry issues described above cannot be modeled by simulation, the Questa® Reset Check app - based on CDC analysis technology - can be employed to automatically perform an exhaustive, bottom-up reset tree analysis, then automatically generates and proves assertions that cover numerous reset-specific structural checks. Using real-world DUTs from customers in a variety of industries, Questa® Reset Check was run on 3 designs of varying complexity including a prototype block, a functional controller and a networking design unit (Table 1 above right). Each of the designs show different interactions between the generated clock-domains and the inferred reset domains leading to domain crossing issues. For example, Design 2 has a large number of reset domains, leading to a large number of reset domain crossings. In general, users can customize the analysis by assigning explicit values to the control signals, or by grouping the clock and reset signals explicitly with directives.

![Figure 8](image)

Other static reset issues found in the designs included:

- Unused user specified resets
- Conflicts between specification and reset usage
- Combinational logic before asynchronous reset synchronizer
- Asynchronous reset used incorrectly in data

The reset domain crossing across data path registers in the same clock-domain is a RDC violation. Figure 9a

<table>
<thead>
<tr>
<th>Design complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of registers</td>
</tr>
<tr>
<td>Number of latches</td>
</tr>
<tr>
<td>Number of RAMs</td>
</tr>
<tr>
<td>Number of Gate-level modules</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reset/Clock Domains Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of reset domains</td>
</tr>
<tr>
<td>Total number of clock domains</td>
</tr>
<tr>
<td>Number of clocks crossing reset domains</td>
</tr>
<tr>
<td>Number of resets crossing clock domains</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of (RDC) Reset Domain Crossings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data path across same clock domain</td>
</tr>
<tr>
<td>Data path across different clock domains</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Results of Static Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing asynchronous reset synchronizer</td>
</tr>
<tr>
<td>Unexpected gate in reset tree</td>
</tr>
<tr>
<td>Reset signal used as asynchronous and synchronous</td>
</tr>
</tbody>
</table>

Table 1: Summary of Reset Verification
shows a simple RDC violation with start register 'mask' reset asynchronously by a signal 'rst' driving the end register 'rx_masked_data' reset asynchronously by a different signal 'clr'. Both the registers are clocked by the output of a scan mode multiplexer thus belonging to the same clock-domain.

Figure 9b illustrates a reset used synchronously and asynchronously in a design. The highlighted path shows the top primary reset input rst_HT_L resetting the first register asynchronously and resetting the second register on the right hand side through a synchronous reset mux select.

**CONCLUSION**

In this article we presented several common reset problems, the corresponding reset verification challenges, and proposed a comprehensive solution to address these challenges. We have also demonstrated the solution’s effectiveness through verification of several customer designs, and presented the issues discovered. We also showed that many of these issues are hard to catch bugs that cannot be caught easily with simulation or traditional verification techniques.

**REFERENCES**


# Debugging Inconclusive Assertions and a Case Study

*by Jin Hou—Mentor, A Siemens Business*

## INTRODUCTION

Formal assertion-based verification uses formal technologies to analyze if a design satisfies a given set of properties. Formal verification doesn’t need simulation testbenches and can start much earlier in the verification process. There are three possible results for an assertion after formal runs: “proven,” “fired,” and “inconclusive.” *Proven* means that formal has done exhaustive mathematical analysis and proved the design satisfies the assertion and no legal stimulus sequence can violate the assertion. *Fired* means that formal has found an error trace showing the assertion violation. *Inconclusive* means that formal has only found bounded proof result, i.e., the assertion is true for the bounded cycle, but may be true or false for longer cycles. An inconclusive result cannot guarantee the design function associated with the assertion is correct.

Considering the size and complexity of designs today, formal users often encounter inconclusive results for some assertions, especially when users run formal on the blocks of designs that have long sequential depths, a huge number of state bits, and complicated mathematic operations. When we get inconclusive results, we can try some simple solutions; such as increasing the run time, running more cores, and using bigger machines. If the assertion still cannot converge, we need to debug the root cause of it. In this article, we discuss the flow to debug inconclusive assertions and use an ECC design as an example to show a decomposition technique for handling inconclusive assertions.

### DEBUGGING INCONCLUSIVE ASSERTIONS

The flow of debugging the root causes of inclusive assertions is shown in Figure 1 (left) and Figure 2 (above, right). When an assertion cannot converge, the user has to check if the design is correctly initialized. Questa® PropCheck reports the registers that are still X or Z after initialization in the generated report “formal_verify.rpt.” The user can

<table>
<thead>
<tr>
<th>Investigate Initial State: open “formal_verify.rpt”</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Look at “X Registers in Starting State” table</td>
</tr>
<tr>
<td>• Look at “Z Registers in Starting State” table</td>
</tr>
<tr>
<td>• Look at initial sequence</td>
</tr>
<tr>
<td>Fix initial state: change RTL, initial sequence, or add “netlist initial”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Investigate Constraints: open “formal_verify.rpt”</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Look at “Port Constraints” table</td>
</tr>
<tr>
<td>• Look at “Assumptions Used in Bounded Proofs” table</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Add “netlist constant” directives (Firings are valid, proofs may be false)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get rid of irrelevant logic (proofs are valid, firings may be false)</td>
</tr>
<tr>
<td>• Use “netlist cutpoint”</td>
</tr>
<tr>
<td>• Use “netlist blackbox” or “netlist blackbox instance”</td>
</tr>
</tbody>
</table>

Target Converges?  

- Success!  
- Goto Flowchart (2)  

*Figure 1: Flowchart (1)*
look at the two tables “X Registers in Starting State” and “Z Registers in Starting State” in the file to find out if the design is properly initialized. If it is not, the user has to fix it, either by changing the design, writing a correct initial sequence, or using the tool command “netlist initial.”

Incorrect constraints may increase formal complexity. Questa® PropCheck reports which ports are constrained and which assumptions are used in the “formal_verify.rpt” file. The user can set some ports to constant, such as setting mode pin to one mode at a time, to reduce the complexity of the circuit. The user can also get rid of some complicated logic by cutting signals, or blackboxing modules or instances. These are easy steps to do. After fixing the initial state, modifying port constraints, or getting rid of some complicated logic, the user can rerun the test. If these simple steps cannot solve the inconclusive issue, the user has to do more analysis on assumptions, assertions, and design logic. Further steps are shown in Figure 2 above.

Sometimes removing assumptions can reduce complexity for formal. If an assumption drags in extra design signals that are not in the Cone of Influence circuit (COI) of the assertion, removing the assumption may be helpful. If an assumption is really complicated, introducing lots of extra state bits, removing it may help. Removing constraints may allow illegal stimulus that can result in false firing. When the assertion is fired, the user can check the error trace generated by the tool to figure out the root cause: either a design bug or missing constraint. If an assertion can be proven with under-constraining, it is still true with more constraints, i.e., the proof is valid. Sometimes adding more constraints can reduce complexity. Reducing the number of values that inputs can get may help. When over-constraining inputs, assertion firing is valid, but proof may be false.

If assertions are not written in a formal-friendly style, performance can be affected significantly. The user may consider rewriting the assertion to reduce state bits and sequential latency. If an assertion is to verify the end-to-end property of a big design, the chance to converge is small. Decomposing an end-to-end property to several simpler properties with small latencies can improve performance a lot. Since Questa® PropCheck can automatically use proven assertions as assumptions for proving other assertions (called assume-guarantee), the user can add helper assertions to leverage the tool’s features to increase performance.
Big memories and counters can introduce a lot of state bits and large cycles for formal to analyze, which can be the root cause of an inconclusive issue. The user can find out the memories and counters used by Questa® PropCheck for a specific assertion by using “formal verify -show_active_logic.” The tool reports the active logic used by formal engines and orders the signals according to their significance. The signal affecting formal the most is listed on top. The user can reduce the size of the memories and counters in the list, or remodel them.

If the user has tried all the steps introduced above, but the assertion still cannot converge, the user can contact Mentor verification support team for help.

**A CASE STUDY**
The design is the ECC logic shown in Figure 3. The encoder generates a syndrome, then the data and syndrome are written to the FIFO, and the decoder reads the data and syndrome from the FIFO and decodes it. The design can correct maximum 2-bit random errors in the data.

The end-to-end property verifies that data can be corrected when errors are less than 3 bits. This assertion may not converge within a reasonable time due to the complicated mathematical operations in the encoder and decoder and the long latency and large state bits introduced by the FIFO. To verify this design, we can verify the encoder and decoder separately from verifying the FIFO. We have to create a wrapper that only instantiates the encoder and the decoder as shown in Figure 4.

After the FIFO is not included, the formal complexity for the property verifying the correctness of the encoder and the decoder is reduced a lot, and the property writing also becomes much simpler. The property can be written using SystemVerilog Assertions (SVA) or the tool command “netlist property” as follows:

**SVA:**

```systemverilog
Check_err_corrected: assert property (@(posedge clk) disable iff (!rstn) dout == din);
```

**Tool directive:**

```systemverilog
netlist property -name Check_err_corrected -assert {dout == din}
```

---

**Figure 3: ECC Design**

**Figure 4: Wrapper for the Encoder and the Decoder**
We can use Questa® PropCheck’s command “netlist cutpoint” to inject random errors using the following tcl file “inject_errors.tcl.”

```tcl
for {set i 0} {$i < 26} {incr i} {
    if {$i < 16} {
        netlist cutpoint data[$i] -cond (randbit[$i]==1'b1)
    } else {
        netlist cutpoint syn[$expr ($i-16)] -cond (randbit[$i]==1'b1)
    }
}
netlist property -name assume_2errs -assume {$countones(randbit) <= 2}
```

Here “randbit” is an undriven wire defined in the wrapper. Formal considers undriven wires as free variables that can get any values randomly. When any bit of “randbit” is 1, the associated “data” bit or “syn” bit is cut. Since we define “assume_2errs” to constrain “randbit” to have maximum 2 bits of value 1, the maximum number of “data” and “syn” bits that can be cut is 2. When a signal is cut, it is not controlled by design logic anymore, instead it becomes a free variable for formal to control. The cut signal bits can be any value; as errors are injected when they get values different from the correct values. The above code can inject 0, 1, 2 errors randomly. When formal proves “check_err_corrected”, it proves “dout” always equals to “din” considering all random 0, 1, 2 errors. The Makefile to run the verification task is as follows:

```
Run: compile formal
compile:
vlib work
  vlog rtl/wrapper.v -y .rtl +incdir+./rtl +libext+.v
formal:
vcover -c -od log -do *
do inject_errors.tcl;
netlist property -name Check_err_corrected -assert {dout==din};
formal compile -d wrapper;
formal verify;
exit*
```

Now we have verified the encoder and the decoder. If the FIFO module has design bugs, the original ECC won’t work. We have to verify FIFO to make sure it is correct. We can use the Questa® QFL library to verify the FIFO. To use the library, we can simply copy the template “qfl_fifo_single_clock. template.sv” under the QFL installation directory to “bind_qfl_fifo.sv,” and modify the parameters and signal names to match the ECC design, shown below.

```
bind fifo qfl_fifo_single_clock
#(.
  .WIDTH  (26),
  .DEPTH  (64),
  .TIMEOUT  (0),
  .ENQ_WIDTH  (1),
  .DEQ_WIDTH  (1),
  .ENABLE_ASSERT  (1),
  .ENABLE_COVER   (1)
)
QFL_INSTANCE_FIFO_SINGLE_CLOCK
  `ifdef QFL_TEMPLATE_PORT_CONNECTIONS_ALL
  *
  `else
  /*input [ENQ_WIDTH - 1:0]                    */   .enq         (push),
  /*input [ENQ_WIDTH * WIDTH - 1:0]   */   .enq_data (data_in),
  /*input [DEQ_WIDTH - 1:0]                     */   .deq        (pop),
  /*input [DEQ_WIDTH * WIDTH - 1:0]    */   .deq_data (data_out),
  /*input */   *.clk  (clk),
  /*input */   *.resetn (rstn)
  `endif

When verifying the FIFO, we can read in the original design ECC to make sure the FIFO is working in the context of the design. For verifying the FIFO, we don’t care about the actual value of the data written into the FIFO. To reduce the complexity, we cut the signals “data” and “syn” to remove the complicated encoder logic. Now “data” and “syn” become free inputs to the FIFO for formal. If the tool can verify that for any values written to the FIFO, the values are read out in the correct order from the FIFO, the data integrity feature of the FIFO is proven. Using the Questa® QFL FIFO library, the tool can automatically verify FIFO data integrity, no overflow, no underflow, and no timeout.

The Makefile to verify the FIFO is as follows. The first “vlog” command compiles the RTL files of the design, the second “vlog” command compiles the QFL library and the bind file. For the FIFO to work
correctly, the primary inputs “wr_en” and “rd_en” have to be constrained such that when the FIFO is empty, “rd_en” cannot be issued, and when the FIFO is full, “wr_en” cannot be issued. We use the “netlist property” command to define these two constraints in the example.

```bash
INSTALL := $(shell qverify -install_path)
run: compile formal
compile:
  vlib work
  vlog rtl/ecc_top.v .y ./rtl +incdir+./rtl +libext+.v
  vlog -sv -mfcu -cuname mybind qft_files/bind_qfl_fifo.sv \ -y $(INSTALL)/share/QFL/checkers +libext+.sv \ +incdir+$(INSTALL)/share/QFL/checkers
formal:
  qverify -c -od log -do " \ 
   netlist cutpoint data; \ 
   netlist cutpoint e_syn; \ 
   netlist property -name constr_no_underflow -assume \ {!(rd_en && empty)}; \ 
   netlist property -name constr_no_overflow -assume \ {!(wr_en && full)}; \ 
   formal compile -d ecc_top -cuname mybind; \ 
   formal verify; \ 
   exit"
```

**SUMMARY**

In this article we discussed the flow to debug inconclusive assertions and also some techniques to reduce formal complexity that may result in inconclusive assertions; including fixing incorrect initial state and incorrect constraints, removing complicated design logic by blackboxing design module-instance or cutting design signals, and reducing assertion complexity. We looked at a testcase where we reduced the complexity of the original end-to-end property by verifying the complicated arithmetic blocks and FIFO separately. There are a lot of creative ways to reduce formal complexity and get more results from formal runs. To learn more, you can check our Verification Academy website for the course “Handling Inconclusive Assertions in Formal Verification.”
Getting Generic with Test Intent: Separating Test Intent from Design Details with Portable Stimulus

by Matthew Ballance—Mentor, A Siemens Business

It’s pretty typical to think about writing tests for a specific design. However, as the number of SoCs and SoC variants that a verification team is responsible for grows, creating tests that are specific to the design is becoming impractical. There has been a fair amount of innovation in this space recently. Some organizations are using design-configuration details to customize parameterized tests suites. Some have even gone as far as generating both the design and the test suite from the same description.

The emerging Accellera Portable Stimulus Standard (PSS) provides features that enable test writers to maintain a strong separation between test intent (the high-level rules bounding the test scenario to produce) and the design-specific tests that are run against a specific design. This article shows how Accellera PSS can be used to develop generic test intent for generating memory traffic in an SoC, and how that generic test intent is targeted to a specific design.

SoCs have complex memory subsystems with cache coherent interconnects for cores and accelerators, multiple levels of interconnects and bridges, and many initiators and targets – often with limited accessibility. While it’s certainly important to verify connectivity between all initiators and targets, it is much more important to generate traffic between the various initiators and targets to validate performance. The goal here is to stress the interconnect bandwidth by generating multi-hop traffic and parallel transfers.

There are, of course, multiple approaches to generating traffic across an SoC memory subsystem. SystemVerilog constrained-random tests can be used, as could manually-coded directed tests. While both of these approaches are tried and tested, both also have drawbacks. Constrained-random tests are generally limited to simulation and (maybe) emulation, but we often want to run these traffic tests across all the engines – simulation, emulation, and prototype. Directed tests, while portable, are incredibly laborious to create, and often miss critical corner cases. A bigger challenge to both approaches, though, is the impact that a design change has on the test suite. Because test intent (what to test) is so enmeshed with the test realization (how to implement the test behavior), a change to the design often results in the need to manually review hundreds (or more) tests to identify needed updates to account for the new things that need to be tested in the design variant, and the things that are currently being tested that no longer exist in the new design variant.

If we take a step back from our memory subsystem traffic-generation problem, our test intent is actually quite simple: generate traffic between available initiators and available targets. Accellera PSS allows us to generalize a surprising amount of our
overall test intent and test scenarios without knowing anything about the specific resources present in the design. Accellera PSS also allows design details to be specified such that they are quite separate from the generic test intent and scenarios, making these easily reusable.

**GENERIC TEST INTENT INFRASTRUCTURE**

As previously stated, memory-subsystem traffic generation involves using initiators to transfer data from memory to memory. We start capturing generic test intent by characterizing a transfer. Specifically, where the data is stored and what initiator is performing the transfer.

Figure 2 shows the Accellera PSS description of a memory transfer and related types. A buffer in PSS is a data type that specifies that its producer must complete execution before its consumer can execute. The `data_xfer_b` type shown above captures the region in which the data is stored, the address and size of the data, what initiator transferred the data, and how many “hops” the data took in getting to its current location.

Figure 2: Generic Description of a Transfer

Note that empty enumerated types have been defined to capture the initiator moving the data (`data_mover_e`) and the memory region where the data is stored (`mem_region_e`). The specific enumerators that compose these types will be specified by the system configuration.

**GENERIC TEST INTENT PRIMITIVES**

Accellera PSS uses the action construct to specify the high-level test intent behavior. The component construct is used to collect resources and the actions that use those resources.

```plaintext
component data_mover_c {
    import data_mover_types_pkg::*;

    abstract action move_data_a {
        input data_xfer_b in;
        output data_xfer_b out;

        constraint in_out_c {
            out.size == in.size;
        }
    }
}
```

Figure 3 shows the PSS description for a generic component action that transfers data. Note that the `move_data_a` action is abstract, which means that it cannot be used on its own (it’s far too generic). However, this generic outline will be used as the basis for design-specific actions that represent the actual initiators in the system.

The `move_data_a` action has an input and an output of type `data_xfer_b`. This specifies that some action must run before an action that inherits from `move_data_a`, and that this action produces a data transfer descriptor that can be used by another action. Figure 4 shows a diagram of the `move_data_a` action with its input and output buffers.

Figure 4: Generic Data Mover Action Diagram

IN

`move_data_a`

OUT
It’s also helpful (and generic) to provide generic actions that produce an initialized data-transfer descriptor and one that accepts and terminates a series of transfers. Basic versions of these can be provided (as shown in Figure 5 above), and more environment-specific versions provided for specific designs.

GENERIC TEST INTENT

With just the infrastructure and primitives we’ve defined thus far, we can already get started specifying test intent. Our first test scenario is shown in Figure 6 below.

```
coverspec point2point_cs(data_xfer_b dst) {
    initiator_cp : coverpoint dst.mover;
    target_cp : coverpoint dst.region;
    initiatorXtarget : cross initiator_cp, target_cp;
}
```

```
action mem2mem_point2point_a {
    data_src a src;
    data_sink_a sink;
    activity {
        src;
        sink with out.num_hops == 1;
    }
    point2point_cs cov(sink.in);
}
```

Figure 7 (below) shows a graphical representation of our point-to-point scenario. Note that, while we have required an action to exist between src and sink, we haven’t specified what it is – just that it must accept an input data buffer and produce an output data buffer. Likewise, our coverage goals are specified in terms of the set of target memory regions and initiators, despite the fact that we don’t know which initiators and targets our design will eventually contain.

**Figure 7: Graphical Representation of Point-to-point**

This ability of a PSS description to capture generic test intent is one thing that makes it such a powerful way to specify test scenarios.

CAPTURING SYSTEM SPECIFICS

Of course, we do have to capture the specifics of our actual design before we can generate interesting and legal tests. Our example system contains:

- Two processor cores
- One DMA engine with 32 channels
- Two accelerators with private memory
- A DDR controller
- An internal RAM and ROM

We need to provide actions to describe the behavior of the initiators in our system that we want to test.

We’ve created a top-level action (mem2mem_point2point_a) that instantiates the data_src_a and data_sink_a actions, and traverses them in an activity block with the stipulation that one action come between (num_hops==1).
Figure 8 shows a PSS component and action that represents one of the processors in our system. PSS groups actions and the resources they require inside a component. In this case, we have a resource type to represent the CPU being used (cpu_r), and a pool of that resource type (core) to ensure that only one CPU operation can occur at one time.

Note that our cpu_c component inherits from the data_mover_c component, and the memcpy_a action inherits from the data_mover_a action.

As a consequence, it will have the same data buffer input and output that the data_mover_a action has.

Figure 9 below shows a description of the DMA component. Just like with our CPU component, we use a resource to describe how many parallel operations can run on an instance of the DMA component. Because we have 32 DMA channels, we create a pool of 32 dma_channel_r resources.

CAPTURING SYSTEM RESOURCES
Thus far, we have captured information about blocks within the design. These components and actions may well have been created by the teams responsible for verifying the IP, and reused at SoC level. Now, though, we need to capture the complete view of the resources and actions available in our SoC.

Figure 10 shows a top-level component with a component instance to capture each available resource in the design. We have two instances of the cpu_c component to represent the two processors, an instance of the dma_c component to represent the DMA engine, and component instances to represent the accelerators.

CAPTURING SYSTEM-LEVEL CONSTRAINTS
Now that we’ve captured the available resources, we need to capture the system-level constraints.

Figure 11 (above right) shows the system-level constraints. Note that we use type extension (the extend statement) to layer our system-level constraints into the existing enumerated types and base action. Like many powerful programming constructs, type extension is very useful when used...
judiciously, though overuse can easily lead to spaghetti code.

In addition to capturing the available memory regions (MEM_codec, MEM_crypto, etc), we also capture restrictions on which initiators can access which targets. Note that we’ve captured the restriction that only accelerators can access their local memories by stating that if either the source or destination memory is the accelerator-local memories, then the initiator must be the corresponding accelerator.

So, all in all, a fairly simple process to capture system capabilities and constraints.

BRINGING TEST INTENT AND SYSTEM SPECIFICS TOGETHER
Now that we have both generic test intent and system specifics available, we can bring them together and start generating specific tests.

Figure 12 shows how we can customize our generic test intent (mem2mem_test_c) with the capabilities of our specific system. Our specific test component extends from the generic test scenario we previously described. By instantiating the mem_subsystem_c component and connecting all the actions to the same pool of buffers, we make our system-specific actions and resources available to our generic test scenario.

Figure 13 shows a few specific scenarios that could result from our point-to-point scenario combined with our system description. One important thing about a portable stimulus description is that it is declarative and statically analyzable. This means that we can use analysis tools to discover exactly how many legal solutions there are to our point-to-point scenario in the presence of the available system resources. In our cases, there are a total of 72 legal point-to-point scenarios.
EXTENDING THE SCENARIO
We can easily expand our set of tests by using the system resource and constraint description we already have, and just altering our original test scenario a bit.

```plaintext
action mem2mem_multi_hop_test_a {
  data_src_a src;
  data_sink_a sink;

  activity {
    src;
    sink with out.num_hops == 2;
  }
}
```

For example, we can alter the number of ‘hops’ our data takes moving from source to sink, as shown in Figure 14. If we increase the number of transfers to 2, there are 864 possible test scenarios. Expanding the number of hops to 4 results in an incredible 124,416 legal test scenarios. Not bad for just a few extra lines of PSS description!

We can just as easily extend the scenario to account for parallel transfers. In this case, we reuse our two-hop scenario and run two instances in parallel (Figure 15).

```plaintext
action mem2mem_multi_hop_parallel_test_a {
  activity {
    parallel {
      do mem2mem_multi_hop_test_a;
      do mem2mem_multi_hop_test_a;
    }
  }
}
```

The resulting transfers will be parallel back-to-back transfers, an example of which is shown in Figure 16. Because we’ve captured the available resources and their restrictions, our PSS processing tool will ensure that only legal sets of parallel transfers are generated.

CHANGING THE DESIGN
Updating a test suite when the SoC changes, or trying to reuse a test suite for an existing SoC on a variant, is laborious and challenging. Just for a start, the set of available resources is different and the memory map is different.

The process is entirely different with a PSS-based test suite. Let’s assume we have an SoC variant that doesn’t have a codec, but does have an additional local RAM (Figure 17).
The only change we need to make is to our description of the system resources. In this case, we need to remove the codec component instance and add another RAM to the memory_region_e enumeration, as shown in Figure 18 above.

With only these minor changes, a PSS processing tool can re-generate specific tests from our high-level test intent that match the new system. In this case, making these design changes expands the number of transfers described by our original point-to-point transfer test from 72 to 128.

**SUMMARY**

As we’ve seen from this simple example, the capabilities of Accellera PSS go far beyond the simple ability to target the same test intent to various verification platforms. PSS allows us to dramatically raise the abstraction level at which test intent is described, allowing us to easily capture generic test intent and test scenarios independent of the design details. Modeling available design resources and constraints and using these to shape test intent is straightforward. Finally, PSS test intent easily adapts to design changes, preserving the effort invested in capturing test intent. Combined, all of these capabilities dramatically boost verification productivity!
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

32 Video Courses Available Covering
- UVM Debug
- Portable Stimulus Basics
- SystemVerilog OOP
- 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 8250 topics
Verification Patterns Library

www.verificationacademy.com