2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
OCTOBER 26-27, 2021

Using HLS to improve Design-for-Verification of multi-pipeline designs with resource sharing

Sarmad Dahir, Cadence Design Systems, Stockholm, Sweden (*sarmad@cadence.com*)
Nils Luetke-Steinhorst, Cadence Design Systems, Munich, Germany (*nls@cadence.com*)
Christian Sauer, Cadence Design Systems, Munich, Germany (*sauerc@cadence.com*)

Abstract

State-of-the-Art verification technologies offer great tools for bridging the gap between design and verification productivity in today's complex design projects. *Design-for-Verification* (DFV) concept invites designers to think and plan ease of verification or "verifiability" early during the design implementation phase. A design, or SoC, implemented with DFV in mind can dramatically improve the verification productivity and increase potential quality of verification.

To satisfy the computational processing demands in today's SoCs it has become quite common to implement multiple pipelines within a single IP. Using multiple pipelines allows for achieving higher throughput but it also increases the complexity of the design. Architecting the access mechanism of shared resources, e.g. memories, in such a design becomes a key factor that affects the overall performance throughput of the design and in some cases the performance of the whole SoC even. This growing design complexity also adds verification challenges in the form of longer time to coverage closure and engineering resources.

Luckily state of the art EDA technologies offers High-Level-Synthesis (HLS) solutions which make the design, analysis, and exploration of different design architectures easier and much faster. Using HLS can also help from the verification aspect. In this paper we discuss how HLS can contribute to better DFV. The input to the HLS tool is an Abstract Behavioral SystemC® model of the target IP. By definition, it has less details and less lines of code. It is also much faster to simulate and easier to debug.

In this paper we will use a design example that implements a memory arbiter which allows multiple pipelines to access an internal on-chip memory without compromising throughput and eliminating stalls on the module IO that could affect the overall SoC performance. We will discuss how verification of this design functionality can be done quickly on the Behavioral SystemC® model and the same TB can then be reused on the generated RTL. This approach can be extended from module/IP level to subsystem or system level verification. When running simulations on system/subsystem level, the instance of the HLS module can easily be switched between RTL and Behavioral SystemC®. We also examined verification features offered by HLS tools such as assertions synthesis and coverage analysis, using different types of coverage metrics on different levels of abstraction, to enable designers and verifiers to discover potential issues in their code much earlier in the design phase.

Keywords— HLS; High-Level-Synthesis; DFV; Design-for-Verification; HW Design; HW Verification; Formal verification; UNR; ASIC; FPGA; RTL; SystemC

I. INTRODUCTION

The complexity of a given design is usually directly related to the number of functional bugs that could be present in it. Complex designs usually include multiple pipelines and large state-machines which makes it difficult to identify and handle all possible conditions and corner-cases by the designer. Capturing such design bugs is also challenging for verification engineers because these bugs could be hidden deep inside the logic and require a very specific set of stimuli to expose them. Another reason that causes design bugs is ambiguous specification. If a system specification doesn't clearly document all possible and valid interface behaviors, then designers would make assumptions on how other modules in the system interact with this module.

Design-for-Verification (DFV) invites designers to think and plan ease of verification or "verifiability" early during the design implementation phase. This would speed-up the verification progress and expose any faulty assumptions or ambiguities. This could be achieved by:

- Using a design flow that models the algorithm in higher level of abstraction → easier algorithm debugging.
- Faster simulations → shorter turnaround time.
- Early coverage analysis capabilities → expose faulty user code or assumptions in both design or verification.
- Ability to combine simulations with other technologies → Make benefit of automated techniques such as advanced formal-verification unreachable (UNR) code analysis.
- Using assertions to highlight illegal scenarios in design code → add clarity on special conditions that would help verifiers identify special corner-cases or faulty assumptions.

In this paper we'll show how the HLS design methodology can aid the DFV approach and help with all the above listed points.

Example design requirements

An ASIC or FPGA design that has multiple input interfaces, which can receive data independently from each other, would be implemented in multiple HDL clocked processes. If the input data, on each interface, could arrive on every clock cycle then it would probably be implemented in a pipelined architecture with initiation interval of one. This way a new data sample can be sampled on the IO interface on every new cycle and pushed into the pipeline to be processed according to the design algorithm. If more than one pipeline needs to access a shared resource, e.g. a single-port memory, then some arbitration logic is needed to control which pipeline gets to access the memory according to a prioritization specification.

For this paper we developed a simple design example using Stratus™ HLS that implements a pipelined Digital-Signal-Processing FIR filter design. The input data is processed in a shift register and Multiply-Accumulate (MAC) operation with input coefficients before being written to an on-chip internal single-port memory buffer. The design also has a second pipeline where a read interface allows upstream modules to request reading the data out from the memory buffer. The operating frequency of this design is 500 MHz and technology library used is a 45nm ASIC library.

To handle the writing and reading of data into the internal memory we implemented a memory arbiter, inside this design, that can receive memory access requests from multiple pipelined threads on every cycle. If more than one pipeline issues a memory access request at a given cycle, the request of the higher priority thread is granted and executed, while the access request from lower priority thread is rejected. In this example the arbiter is also required to signal back the status (ACK or NACK) of each memory access request to the requesting thread.

The input data processed by the filter logic has the highest priority to be written to the internal memory, while the read interface has lower priority. The following figure shows a block diagram of this design.

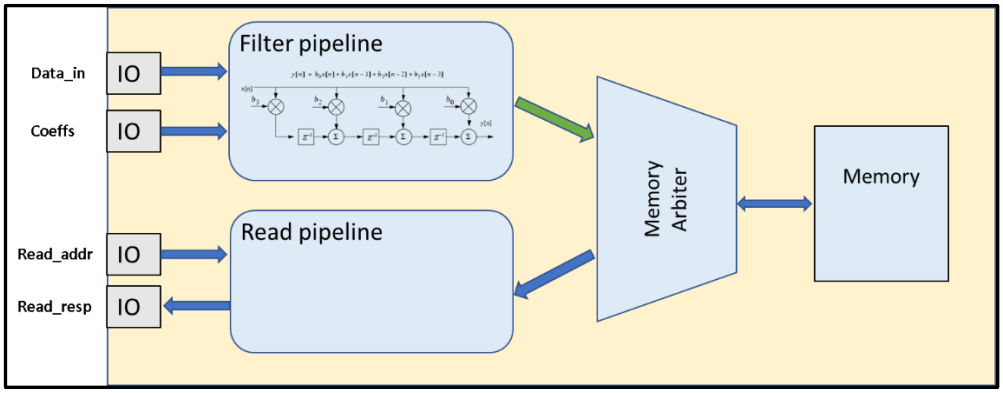

Figure 1: Design block diagram

When implementing such a design using Stratus™ HLS, each of the two pipelines could be modelled as a separate SC_CTHREAD. The desired memory arbiter functionality needs to be modelled in the design code as an additional pipelined thread.

As the desired functionality specification demands that the arbiter to signal back to all threads if their memory access requests were granted or rejected, then a backwards path is needed to acknowledge the status of the memory access request. A simple mechanism for communicating between the threads is to use FIFOs. In this case we'll add Request & Acknowledgement pairs of FIFOs for each pipelined thread accessing the memory. Each pipelined thread will put its memory access request in its own Request FIFO. The memory arbiter thread gets the requests from all Request FIFOs, handles them by either accessing the memory or rejecting the request, and then puts the responses in the Acknowledgement FIFOs. Then the requesting thread gets the acknowledgement response from its own Acknowledgement FIFO to check if its memory access request was executed or rejected. The following figure shows how the pipelined threads communicate through the FIFOs.

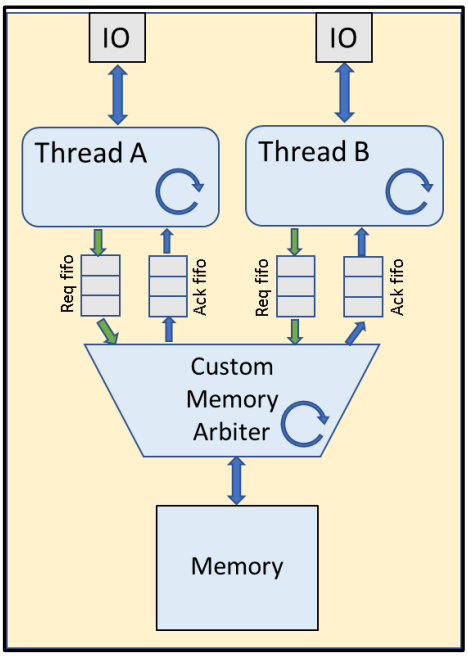

Figure 2: SystemC Design architecture

II. PROBLEM DEFINITION

When verifying such a design the first task is making sure that the design specification is captured correctly in the design implementation, and exhaustively test all *reachable* corner-cases. All *unreachable* corner-cases needs to be analyzed to determine if they are design defects. This can be a time and resource demanding task depending on the complexity of the design algorithm, and the way the design is architectured and implemented. Starting with a design methodology that enables DFV can simplify the verification phase and offers better quality of verification.

A possible first solution is to execute the algorithm verification on a higher level of abstraction; on the Behavioral SystemC® model because it is easier to debug and it's faster to simulate, and then reuse the verification code on the generated RTL. To make benefit from this approach the Behavioral SystemC® model needs to match the produced RTL model IO layout and also match in simulation timing behavior. By achieving this it would be possible to instantiate either model in the TB and reuse the verification code.

Writing the Behavioral SystemC® model in a way that makes it match the generated RTL in simulation timing behavior is not always straightforward because of the difference in nature between event-based RTL simulation, and Behavioral SystemC® simulation. The input SystemC® HLS code can be written in cycle accurate style, but it

would be so low-level and comparable to RTL coding style, and thus the main purpose of using HLS is lost. When using HLS, the designer aims at writing high-level abstract behavioral code with focus on the algorithm, to gain productivity, and lets the HLS tool care about the low-level implementation details.

In an RTL implementation you have a real pipeline where the thread loop iterations are running concurrently, so you start a new loop iteration every new cycle (if initiation interval is one). In Behavioral SystemC® there is no pipelining. The loop iterations are running sequentially. Each get() will consume 1 cycle, the following operations will execute at zero time followed by the put() that also consumes 1 more clock cycle. The full loop iteration must finish before starting the next iteration. This difference in simulation nature usually creates timing mismatches between RTL simulations and Behavioral SystemC® simulations.

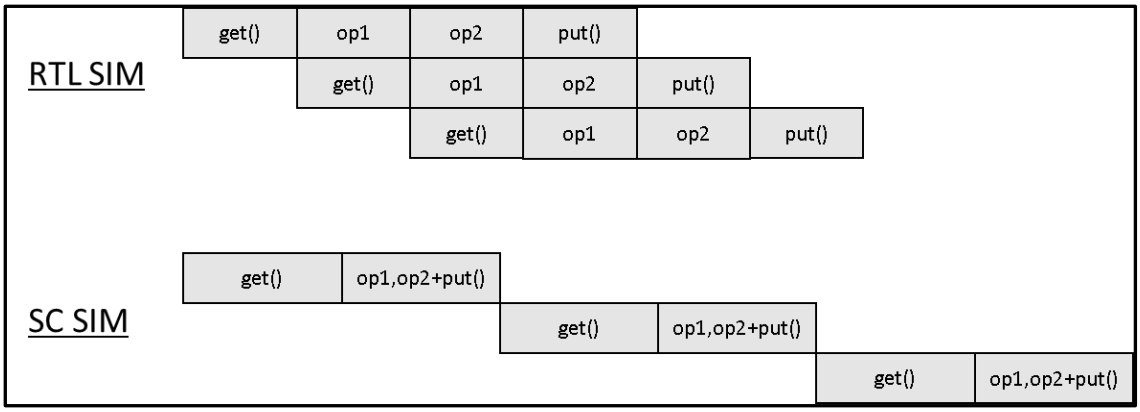

Figure 3: Difference in simulation timing behavior between pipelined RTL and Behavioral SystemC

To make these two design representations match in simulation timing, the Behavioral code needs to be written slightly differently to remove the extra unnecessary wait-cycles in each iteration and be able to receive new input data on every new cycle.

Using assertions in the design code is a very efficient way for designers to highlight illegal conditions, as per their understanding of the specifications. These assertions would also guide the verification engineers to special corner-cases that could expose ambiguous specification or faulty assumptions. When using HLS, the designer can add SystemC/C++ assertions in the input SystemC® behavioral code, but we would need these assertions to be implemented in the generated RTL code as well to highlight the same conditions. This would help the verifiers make benefit of the assertions even when running simulations on the target RTL representation. This requires the SystemC/C++ assertions to be synthesized into the produced RTL implementation.

Using code coverage analysis to identify untested code segments can dramatically improve the verification quality. Designers and verifiers can apply code coverage analysis on both the behavioral SystemC® and RTL representations. The RTL code coverage analysis can be augmented with advanced formal-verification unreachable code (UNR) analysis technologies to identify potential improvements in the verification code or possible issues in the design itself. Designers can use the UNR data to identify any unreachable code that could be a design defect, and verifiers can use this data to identify and target the reachable but uncovered code segments. To make full benefit of RTL code coverage analysis data we need to be able to trace back the uncovered RTL code lines to the originating behavioral SystemC® code lines. This RTL-to-SystemC "traceability" needs to be available to both designers and verifiers so debugging can be done efficiently on the abstract behavioral SystemC® code.

III. PROPOSED SOULTIONS

When using Stratus™ HLS tool, the input is synthesizable behavioral SystemC® code. The IO interfaces of the module can be implemented in both TLM and PIN accurate representations. The proposed solution is to make benefit of this abstract behavioral SystemC® model and do the algorithm verification on this level and then reuse the verification code, the TB and test cases, on the RTL representation. Doing verification on the behavioral SystemC® model is fast because it's more abstract and easier to debug, and it's faster to simulate as well. The selection between instantiating either the behavioral SystemC® or RTL DUT representation is done at compile time.

No changes are required in the module connections because both behavioral SystemC® and RTL DUT representations have matching IO interfaces. Using the behavioral SystemC® model in simulation can be applied to system/subsystem level simulations as well. The HLS design in either RTL or Behavioral representations can be instantiated inside a SoC without breaking the integration. This allows for simulation speed up in the system/subsystem verification phase. To achieve this, we have two requirements:

1. The IO interfaces layout of both RTL and Behavioral SystemC® models must match.
2. The timing/latency behavior on the IOs needs to match to avoid losing data samples.

To achieve matching simulation timing behavior, we need to write the SystemC® code with some considerations. We need to make the *put*() and *get*() function calls non-blocking to avoid stalling the pipelines. Before calling the non-blocking functions, we need to check the IO interfaces and internal FIFOs to determine which has data to transfer. The *nb_get*() function is then called on the interface or FIFO that has a pending item to deliver. Stratus™ HLS tool provides the designer with a variety of different APIs that can be used to model any behavior. In this case we need to *poll* on the IO interfaces and FIFOs. To accomplish this, we can use the *cynw_poll_any*() function which will examine the ports passed to it as arguments, in 1 clock cycle, and then return an array of flags that indicate which inputs have valid data to transfer.
The following figure shows pseudo code that uses FIFOs and non-blocking nb_put() & nb_get() functions to communicate with the other threads and IO interfaces.

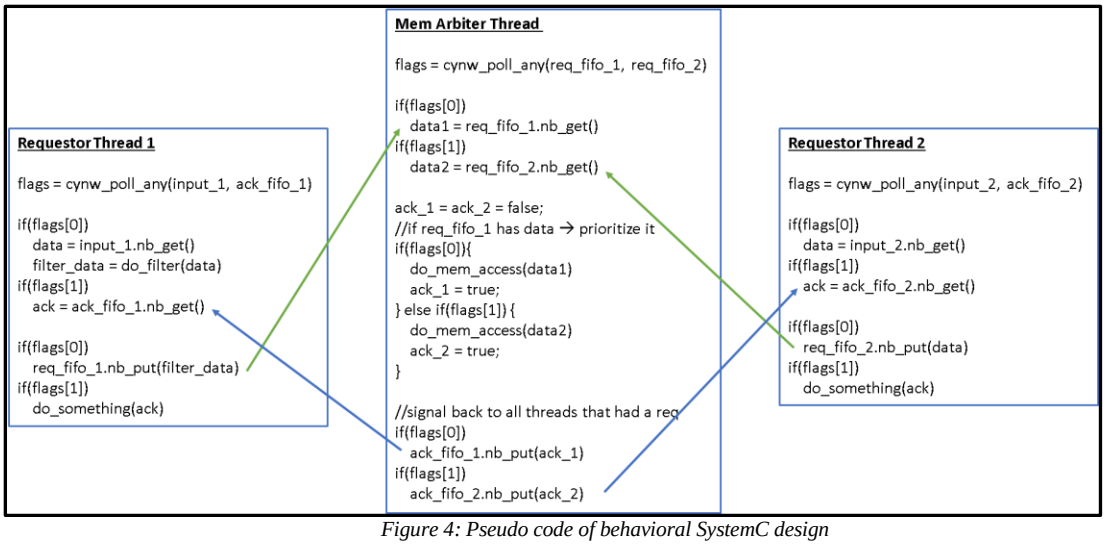

Figure 4: Pseudo code of behavioral SystemC design

Writing the Behavioral SystemC® code this way will give correctly pipelined RTL and gives matching Behavioral SystemC® simulations as well. This opens the door for using either the cycle accurate RTL model or faster Behavioral SystemC® model when executing verification.

Another way to improve DFV is for designers to add assertions in their code to identify illegal conditions. For example, the high-priority thread should never get a NACK response on a memory access request. Using Stratus™ HLS, SystemC/C++ assertions can be synthesized into the generated RTL implementation by turning on the *synthesize_asserts* feature. This allows the designer to communicate effectively and clearly the design intend and assumptions to the verifiers. The verifiers would use these assertions as targets to be verified and covered in their testbenches. Assertions are clear coverage analysis points that can be reviewed using assertion coverage features in simulation. The following figure shows an assertion in the input behavioral SystemC® design, and the equivalent Verilog assertion synthesized into the produced RTL, and also the assertion coverage report that confirms that this design feature was exercised and verified by the verification test cases.

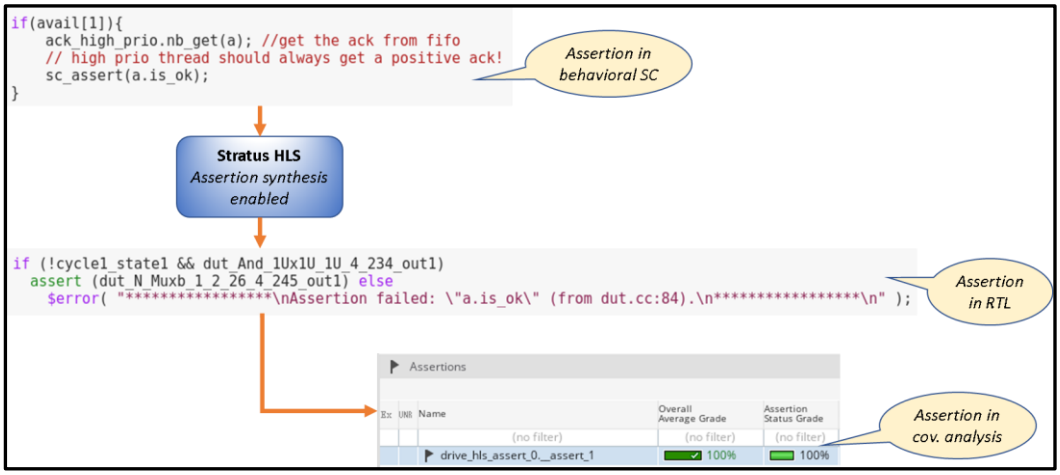

Figure 5: Assertion synthesis flow

To aid verification coverage closure and achieve better DFV, Stratus™ HLS offers integrated SystemC® code coverage (gcov) collection and analysis. As an example, our TB achieved 98.9% line coverage, where one single line was not covered in the Behavioral SystemC® code.

Figure 6: SystemC code coverage report summary

By looking at the annotated coverage report inside Stratus™ IDE, we could see that the highlighted coverage hole is at line 181 because the TB never executed a memory read request through the high-priority thread. This could be a shortcoming in the verification test case, or a design error if the design is only supposed to do write accesses to the memory through the high-priority thread.

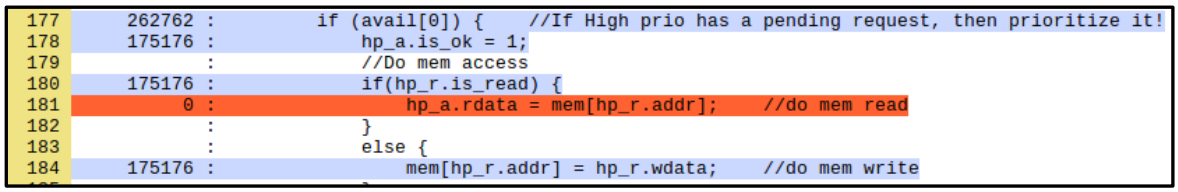

Figure 7: SystemC code coverage report inside Stratus IDE

Another approach to improve verification coverage closure even further was proposed in [1] where JasperGold™ UNR App was used to automatically and exhaustively explore the reachability of yet-to-be-hit cover points in the RTL simulation coverage database. This basically analyzes and identifies reachable and unreachable code coverage holes. The reachable coverage holes are usually potential improvements to the verification environment itself. Unexpected reachable or unreachable coverage holes in the design expose either potential bugs in the DUT or parts that should be excluded from the coverage analysis.
By turning on the *rtl_annotation* feature in Stratus™ HLS, the uncovered RTL code lines are easily traced back to the originating behavioral SystemC® code lines. This saves weeks of verification effort by quickly identifying weaknesses in the TB and/or possible bugs in the DUT, and also by allowing developers to debug the involved logic on the higher abstraction behavioral/algorithmic SystemC® model.

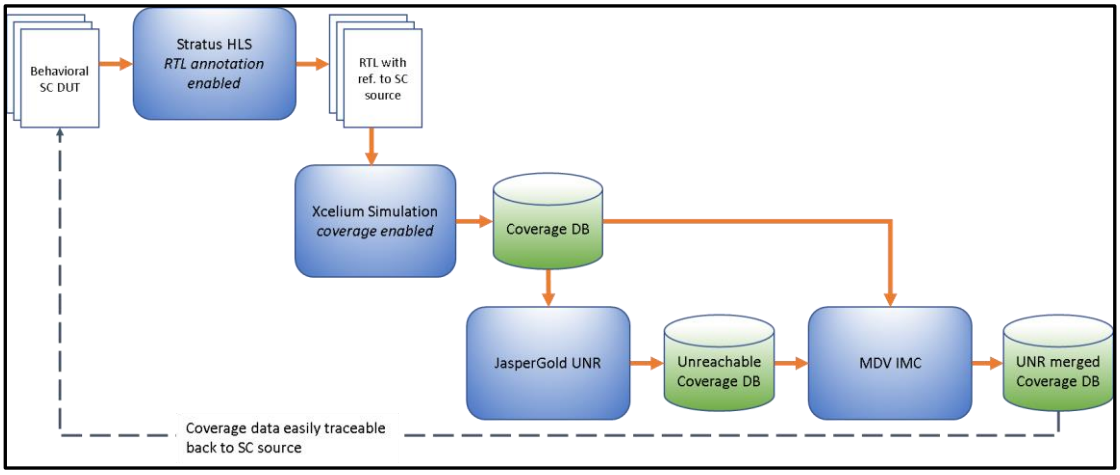

Figure 8: UNR analysis flow with traceability back to SC source

This approach was applied to our example design and it gave few interesting pointers to corner-cases in the pipelines that should be verified to achieve higher confidence. The following picture shows the case where the RTL code that handles a stall due to output backpressure (TB not ready to receive data) was not tested. The inline comments added by the HLS tool in the generated RTL code show the call stack and SystemC® originating code line. The following figure shows the uncovered RTL code block and the direct reference to the originating SystemC® code.

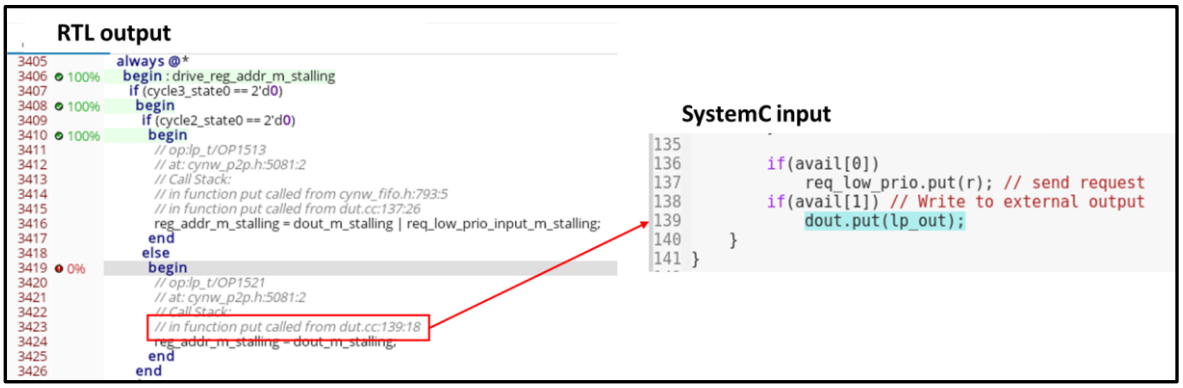

Figure 9: RTL annotation for traceability back to originating SystemC code lines

IV. RESULTS

When reusing the SystemC® TB and running the same test cases on both RTL and behavioral SystemC® representations of the DUT we can see more than 3X simulation speed up with the behavioral SystemC® version compared to RTL simulation. System memory consumption peak in Behavioral SystemC® simulation is 80% compared to RTL simulation.

The number of code lines in Behavioral SystemC® model is only 3,57% compared to the generated RTL representation which makes it much easier to debug.

Enabling SystemC® code coverage exposed untested major features, or potentially incorrect design assumptions to the designer in a matter of seconds.

It was easy and straight-forward to integrate with JasperGold™ UNR formal verification engine that gave more insight on untested low-level RTL features. The traceability back to originating SystemC® code lines made it very trivial to understand what needs to be improved.

V. CONCLUSIONS

The behavioral SystemC® design input to HLS can easily be written in a way so it matches RTL simulation timing behavior. This enables faster verification by using these SystemC® models to speed up simulations on module, subsystem, or system level.

Enabling SystemC® code coverage is a very easy and effective way to analyze the DUT. It can quickly expose untested major features or faulty design specification assumptions.

The ability to combine simulations with UNR formal verification technologies gives the developers a clear view of untested low-level RTL features, and the unexpected (un-)reachable coverage holes could expose DUT bugs. Using the Stratus™ HLS *rtl_annotation* feature we can easily trace back the uncovered RTL code lines to their originating SystemC® code lines and examine the design in its abstract behavioral version.

Using the Stratus™ HLS tool to synthesize SystemC/C++ assertions allows the designer to highlight illegal conditions in the input behavioral SystemC® design and the same assertions would be implemented by the tool into the produced RTL code. This would make these illegal conditions very clear for verifiers and help them identify target corner-cases to be verified.

The HLS methodology helps bring verification awareness early into the design phase and hits all the points mentioned earlier in the introduction section that contribute to better DFV:

- Using a design flow that models the algorithm in higher level of abstraction → easier algorithm debugging.
- Faster simulations → shorter turnaround time.
- Early coverage analysis capabilities → expose faulty user code or assumptions in both design or verification.
- Ability to combine simulations with other technologies → Make benefit of automated techniques such as advanced formal-verification unreachable (UNR) code analysis.
- Using assertions to highlight illegal scenarios in design code → add clarity on special conditions that would help verifiers identify special corner-cases or faulty assumptions.

References

[1] K. Thottempudi, “Coverage Closure for HLS based design IP”, Qualcomm Technologies, 2019-04-02.