Automated Cross-level Verification Flow of a Highly Configurable RISC-V Core Family with Custom Instructions

Stanislaw Kaushanski, MINRES Technologies GmbH, Duisburg, Germany (stas@minres.com)
Eyck Jentzsch, MINRES Technologies GmbH, Munich, Germany (eyck@minres.com)

Abstract

As the RISC-V ISA continues to gain popularity and adoption, there is a growing need for highly configurable RISC-V cores that can be customized for specific use cases and applications. However, the complexity and variability of these cores make verification a significant challenge, especially when it comes to verifying custom instructions. In this paper, we present a cross-level [1] test environment to verify all cores with custom instructions and various interfaces without manual adjustments of the test environment, which significantly reduces the time and effort required for verification.

Introduction

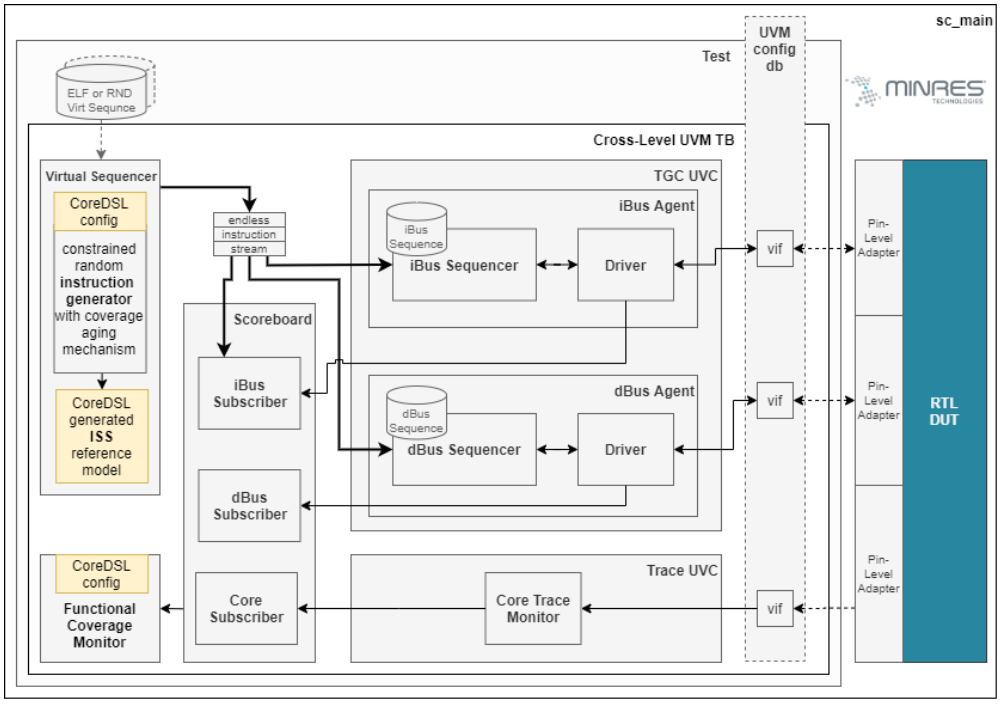
The RISC-V instruction set architecture (ISA) has gained widespread adoption due to its open-source nature, configurability, and scalability. As the RISC-V ecosystem continues to grow, it becomes increasingly important to have robust verification methodologies to ensure the quality of the designed RISC-V cores.

The MINRES TGC core family consists of a set of basic configurations with varying pipeline stages, registers and instruction sets. Each basic configuration can be customized and extended with additional features such as different bus interfaces, interrupt controllers, memory protection, caches and custom instructions.

This high level of configurability in conjunction with functional safety requirements implies a highly automated, modular, and reusable verification environment.

To address this challenge, we use a CoreDSL [2] based approach. CoreDSL is a domain-specific language that provides a unified representation of the RISC-V ISA and its extensions. From a CoreDSL based ISA specification we generate artifacts used in our cross-level testbench environment. This environment employs a constrained-random generator to produce an endless stream of instructions which are executed on both the ISS and RTL models of the core. The behaviour of the ISS is used as a reference to verify correct functionality of the RTL model.

In the following section, we describe the design and implementation of the cross-level testbench, and provide details on the methodology used to generate and transfer instructions, as well as to monitor and compare the behaviour of the ISS and RTL implementations.



Methodology

The verification flow starts with description of the core in CoreDSL as single-source-of-truth. From this description, we generate the instruction accurate and cycle approximate ISS implementations, as well as the configuration for the instruction generator and functional coverage monitor.

Fig.1 shows the structure of the cross-level testbench. The reference generator module combines the ISS and instruction generator to create the reference stream for simulation. When the simulation starts, the ISS fetches instructions, and the fetch accesses are served by the instruction generator. This generator randomly selects instructions from the list of available instructions generated from the CoreDSL description and passes the instruction to the ISS by responding to the fetch access.

The coverage aging mechanism implemented in the instruction generator tracks the randomly selected instructions and regulates their generation frequency to achieve sufficient functional coverage for each instruction while avoiding unnecessary repetitions.

The ISS generates memory accesses on the data bus depending on the executed instructions, and an ISS plugin provides insight into the ISS state and delivers information on exceptions, jumps, branches, and more. Both memory accesses and the internal state changes are recorded in the reference generator and stored in a structure that contains all the necessary information about the current instruction. The structure is then passed on to the UVM agents, which forwards it to the DUT as well as to the scoreboard for verification of the DUT behaviour.

The cross-level UVM TB allows flexible and re-usable simulation of different hardware designs. The UVM-SystemC methodology and standard [3] combines the powerful features of UVM, a state-of-the-art hardware verification methodology with SystemC, which significantly simplifies the integration of the ISS, written in C++.

The UVM-SystemC verification environment is composed of several key components, including agents. Agents are responsible for interfacing with the DUT. The testbench has an iBus and a dBus agent. Each of these consists of a sequencer and a driver. The sequencer acts as the central component that manages and controls the flow of the stimulus to the DUT, ensuring that the testbench applies stimuli in the correct order and at the right time.

During simulation, the DUT initiates instruction fetches. These fetch accesses are received by the iBus driver through a virtual interface (vif). The vif is a connection between the driver and the DUT, that allows them to communicate without being bound to a specific implementation. This means that the DUT can be replaced with different implementations without changing the interface itself.

All cores of the TGC family have an interface for debug and trace purposes. The trace interface maps the internal state of the core with each clock cycle, which is then read by a UVM monitor. The monitor forwards the information to the scoreboard, where it is compared with the state predicted by the ISS, reporting any discrepancies that are found. If such a discrepancy is detected, an error is reported, the testbench dumps a list of last instructions before the error occurred and the verification process is stopped.

Among other checks the scoreboard is also responsible for verifying the correctness of memory access. Specifically, it ensures that the DUT core is accessing the expected address, the access type (READ or WRITE) matches with the ISS, and performs other relevant checks.

Each instruction that is successfully verified in the scoreboard is sent to the coverage monitor. The coverage monitor is a key component of the verification environment that is responsible for collecting and reporting functional coverage data. It is based on FC4SC [4] which is currently in the process of standardization at Accellera.

The coverage monitor defines cover groups for each instruction type, which include coverpoints that track all parameters of an instruction and combinations of parameters within an instruction (crosspoints), as well as hazards between instructions. These cover groups are instantiated for each instruction in the CoreDSL description. They are used to identify areas of the design that require further testing to ensure complete functional coverage.

Conclusion

Overall, the UVM-SystemC cross-level verification environment presented in this paper offers a highly effective solution for detecting a range of design issues, including control and data hazards. This approach is not limited to the TGC core family and can be applied to other highly configurable RISC-V core families. While this particular testbench and TGC core family is used as an example, the methodology and tools described here are general enough to be applied to other core families with similar levels of configurability.

The modular design of the testbench ensures that new core configurations and custom instructions can be added seamlessly, without disrupting the existing verification environment. The use of UVM-SystemC provides a scalable, modular, and reusable solution that can be executed on different simulators, including Verilator, Xcellium, or even on an FPGA e.g. using Raven.

References

[1] V. Herdt, D. Große, E. Jentzsch and R. Drechsler, "Efficient Cross-Level Testing for Processor Verification: A RISC- V Case-Study," *2020 Forum for Specification and Design Languages (FDL)*, Kiel, Germany, 2020, pp. 1-7, doi: 10.1109/FDL50818.2020.9232941.

[2] CoreDSL2 documentation: a domain specific language to describe instruction set architectures https://minres.github.io/CoreDSL/

[3] Stephan Schulz, Thilo Vörtler, Martin Barnasconi, “UVM goes Universal – Introducing UVM in SystemC” DVCoN Europe 2015

[4] D. Dospinescu, T. Vasilache, “Functional Coverage for SystemC (FC4SC)” SystemC Evolution Day 2018