A core challenge

A common verification methodology available to both RISC-V core providers and SoC teams integrating these cores is required, argues **Nicolae Tusinschi**

Modern processor designs present some of the toughest hardware verification challenges. Verification is particularly challenging for RISC-V processor core designs, with many providers and many variations of implementation.

The flexibility that RISC-V provides makes it especially difficult to verify. First, the defined instruction set architecture (ISA) has many optional features and possible variations. The processor has 32 registers that can be 32-, 64-, or 128-bit. The baseline I instruction set has an optional E version that supports only 16 32-bit registers for embedded applications. Additional instructions that may be supported include:

- M extension for integer multiplication/division;
- A extension for atomic read-modify-write memory accesses;
- F extension for single-precision (32-bit) floating point;
- D extension for double-precision (64-bit) floating point;
- Q extension for quad-precision (128-bit) floating point;
- C extension for compressed (16-bit) instructions.

The optional floating-point instructions add 32 more registers of appropriate width. The RISC-V ISA defines three privilege levels, nine exceptions associated with privileged instructions and 4096 control and status registers. Just checking for compliance to the ISA is a significant challenge. But full verification of a RISC-V core for functional correctness goes beyond compliance. The ISA allows the definition of custom instructions and these must be verified to ensure that they work correctly without breaking compliance of the standard instructions. The RISC-V ISA was designed to map to many different microarchitectures, from small controllers to multi-core implementations with the most advanced processor features. A core provider must be able to verify the entire design, including the microarchitectural details, not just ISA compliance. A core integrator may wish to repeat some or all of this verification as an acceptance test. Finally, any register transfer level (RTL) design must be statically analysed to eliminate common coding errors.

A methodology meeting all the requirements for verification of RISC-V cores has to be based on formal technology because any approach based on simulation or emulation can explore only a tiny percentage of design functionality and there is no way to be sure that constrained-random or hand-crafted tests find all the hardware bugs.

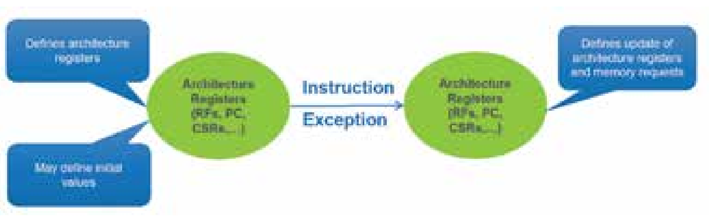
Why formal verification

Only formal verification can prove the absence of something, in this case parts of the design that might pose risks for end applications with high security or trust requirements. Such a formal-based methodology has been developed and deployed on multiple RISC-V core designs. This methodology spans functional correctness, security and trust for RISC-V designs, and can be run by both core providers and core integrators.

Inputs to the process include the core RTL, the ISA compliance rules captured formally and input from the core provider on design decisions such as the number of pipeline stages, which enables verification of the microarchitecture to support the full range of RISC-V verification.

The heart of the methodology is the formalisation of the RISC-V ISA as a set of SystemVerilog assertions using an operational assertion library. This approach defines high-level transactions concisely, similar to timing diagrams to capture the expected results for processor instructions.

Each instruction in the RISC-V ISA is captured in a single operational assertion that applies to any microarchitectural implementation of the instruction. Formal engines verify that the processor state at the end of each instruction's execution matches the specification. This formal approach finds all bugs related to functional correctness and then proves that no further bugs exist. It is flexible enough to handle user extensions beyond the ISA (Figure 1).


Defines architecture registers
May define initial values
Architecture Registers (RFs, PC, CSRs,...)
Instruction
Exception
Architecture Registers (RFs, PC, CSRs,...)
Defines update of architecture registers and memory requests
Figure 1: Operational assertions applied to RISC-V ISA

The formal engines verify the complete RISC-V core beyond ISA compliance, including the microarchitecture. The methodology also calls for running static analysis on the core to find common design errors, from simple syntax mistakes and typographical errors in the RTL to weaknesses that could compromise the final chip. This process is fully automated, so many design teams set it to run on every attempt to check RTL code into a revision-control system.

Some applications for RISC-V processors have strict security requirements so that malicious agents cannot exploit vulnerabilities in the design. Security verification must include a rigorous process to detect all bugs and flaws, establishing precisely what can and cannot happen in the design.

The formal-based RISC-V verification methodology includes running a wide range of automated checks on the core RTL design, rapidly eliminating many classes of common coding and design errors. These checks include: structural analysis (syntactic and semantic analysis of source code); safety checks (exhaustive verification of the absence of common sequential design operation issues); and activation checks (proof that all design functions can be executed and are not blocked due to unreachable code).

Some of the problems uncovered by these checks represent security risks. For example, an incomplete case statement leaves unspecified what will happen if an unexpected value occurs. Such issues can be found and fixed automatically early in the design process.

Malicious code

Inadvertent vulnerabilities are worrisome, but deliberately inserted malicious code (hardware Trojans) is an even bigger concern for trust-critical applications such as autonomous vehicles and nuclear power plants.

Problems may creep in during logic synthesis, either by deliberate action or tool issues. Integration of cores such as RISC-V processors presents extra risk because the SoC team does not know the details of those designs. Tool bugs or intentional netlist modifications can occur during the place and route process. The RISC-V verification methodology includes two steps to ensure trust in a processor core and extend to any SoC integrating it.

Trust in the design itself relies on formal verification's ability to detect when a design can do something that it is not supposed to do. Beyond verifying compliance to the RISC-V ISA, the methodology ensures that the set of assertions covers the entire core design.

GapFreeVerification technology detects and reports any specification omissions and errors, holes in the verification plan, and unverified RTL functions. Any unexpected functionality is flagged, since it could potentially be a hardware Trojan inserted during the design process. Core providers want their products to be trusted and core integrators may want to verify that trust themselves.

For the remainder of the development flow beyond the RTL stage, the RISC-V verification methodology uses formal equivalence checking to ensure that no new logic is introduced during logic synthesis or place and route. This process also ensures that the aggressive optimisations available during implementation are appropriate for the design and have not altered it in some unexpected way. Verifying post-synthesis and post-route netlists against the input RTL specification proven correct ensures trust in every stage of the core and SoC development process.

The verification methodology described has been deployed at multiple sites and runs on multiple RISC-V processor core implementations. Results for two cores available from open-source repositories can be shared publicly.

The first is RI5CY, a 32-bit implementation with a four-stage in-order pipeline. It supports the IMFC instruction sets plus customer instruction set extensions intended for signal processing. The RISC-V verification methodology has identified 13 issues to date, which have been reported back to the RI5CY developers, who have confirmed four issues as bugs and fixed three. The remainder are still under investigation.

The second design verified by the methodology is Rocket Core, a 64-bit implementation of RISC-V with a five-stage, single-issue, in-order pipeline. It has a sophisticated microarchitecture with branch prediction, instruction replay and out-of-order completion for long-latency instructions such as division. Five issues were found and reported to the Rocket Core developers.

One bug where the core contains an undocumented non-standard instruction was reported by the formal engines. The unexpected instruction could have been a hardware Trojan but it turned out that the development team had added a new instruction but had not yet documented it in the specification that the verification team used to develop operational assertions for the custom instructions. Any integration team that downloaded the same version of the core would have had unknown and undocumented functionality in the design, but GapFreeVerification could have detected and reported this. ❑

About the author

Nicholae Tusinschi, product manager, OneSpin, a Siemens business

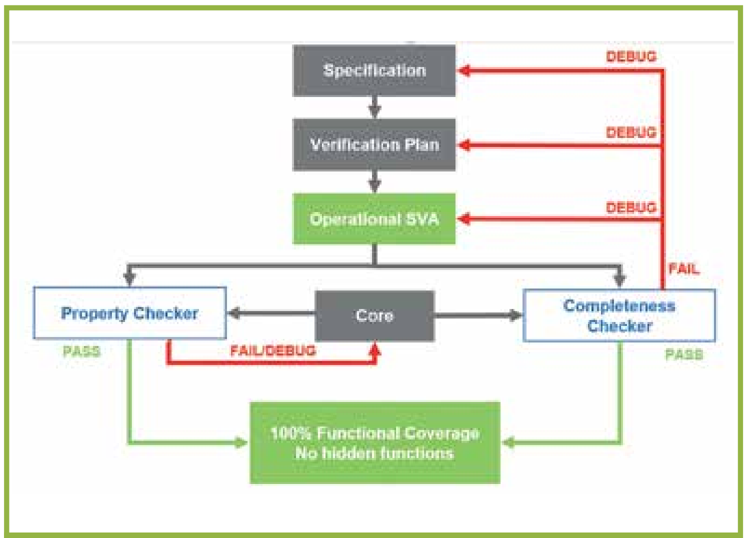


Figure 2: GapFreeVerification flow

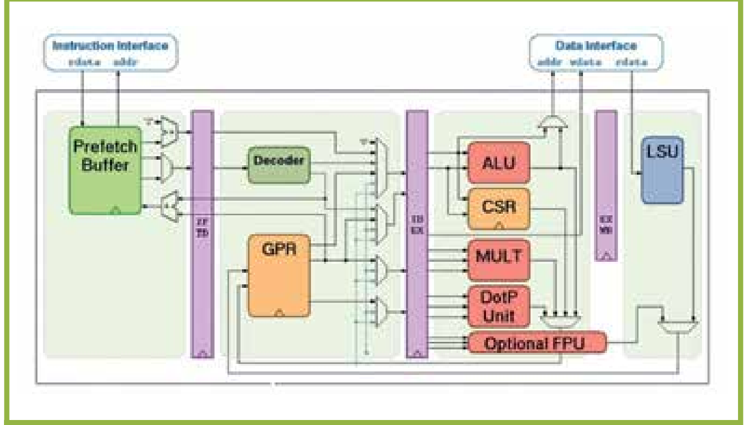


Figure 3: RI5CY block diagram

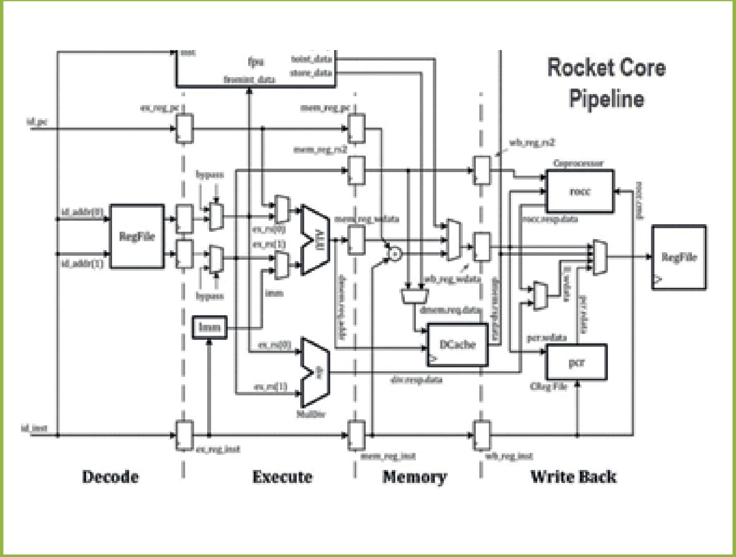


Figure 4: Rock Core block diagram