CONVERTER MAGAZINE
Volume 2021, No. 7

Formal Verification in RISC-V CPU Design Verification Flow

Xueying Yang, Yongzhong Zhou, Lei Li, Weili Li, Liang Liu, Yichu Jiang
Beijing Smart-Chip Microelectronics Technology Co., Ltd., Beijing, 100192, China

Abstract

In recent years, the popularity of RISC-V cores has continued to rise. The flexibility and scalability of its architecture have brought convenience in applications, but it has also brought huge challenges to design verification. For traditional simulation based verification, proof of correctness and security is hard to achieve. At this moment, formal verification with its advantages in completeness and correctness behavior has become a new choice. It has become improvement for simulation based verification. Even more, when a full proof is done, it can replace simulation, and ensure the correctness of the design. This article introduces the theory and actual implementation of processor formal verification.

Keywords: *Formal verification, RISC-V, Formal Spec*

I. Introduction

Since the open source instruction set architecture RISC-V was introduced in 2010, its members have exceeded 1,000. Because of its free, simple and easy-to-use features, diverse configurations and easy expansion, a rich ecosystem has been developed in a short period of time. There are more than 100 RISC-V cores released on the official website list. It ranges from simple cores for educational purposes that only support the most basic instruction set RV32I, to high-performance, multi-core systems. How to verify the correctness, security, and credibility of the core functions of these micro-architectures is a huge challenge for verification.

The traditional random simulation verification method based on UVM is difficult to give satisfactory answers to the efficiency and coverage of verification excitation. And the scenarios of the processor are complex, how to ensure that the instructions are executed correctly under all corner cases, and at the same time, do not behave as not expected, which is difficult to achieve for simulation based verification. Formal verification is the best choice for the completeness and correctness of verification due to the characteristics of its methodology.

II. Advantage of Formal Verification and a Real Case in CPU Verification

In [1], formal verification is defined as follows, *FV is the use of tools that mathematically analyze the space of possible behaviors of a design, rather than computing results for particular values*. Formal verification is the process of using mathematical methods to verify the correctness of a design. Its tools use various algorithms to verify the design, but do not perform any timing checks. The most fundamental difference between formal verification and simulation is that formal verification is not based on some given input vector, but through mathematical analysis and derivation. It is to prove whether a certain logic function is fully consistent with the design specification within the given boundary range, and if not, a counter-example will be given.

2.1 Advantage of formal verification

For some large-scale designs, even if multiple different inputs are used, it is still possible to find the same or partially overlapping paths, which makes it difficult to cover certain states, which is usually called "corner cases".
To deal with corner cases, the usual methods include: increasing the number of simulations, that is, trying more different input combinations; extending the simulation time; or creating a new "directional test" for it. However, no matter what method is used, it will greatly increase the time required to complete the verification and the various

costs invested.

The most fundamental problem is that even if these methods are used, there is no guarantee that the corner cases will be 100% covered. This is because the simulation cannot traverse all possible input vectors. In other words, simulation-based verification only verifies that when certain test vectors are used, the system will not have vulnerabilities, but it cannot guarantee that when other test vectors are used, vulnerabilities will not appear.
Compared with traditional simulation, simulation or other verification processes, formal verification has many advantages:

➢ Solve the problem of correctness: verify that the hardware not only does what it should do, but also does not do what it should not do.

➢ Complete coverage: For possible design behaviors, formal methods can provide complete coverage.

➢ Generate the smallest example: The formal engine is essentially capable of generating the smallest example of the required behavior, but in a typical random simulation environment, it will be embedded in thousands of cycles of random operation.

➢ Boundary situations: The formal method allows any behavior that is not explicitly excluded, which means that the tool is likely to find extreme situations that the user would not think of.

➢ State-driven and output-driven analysis: Because formal tools allow to constrain logical behavior at any point in the design, including internal state or output, it enables us to reason that the behavior of the design can or cannot lead to a specific state or output of interest.

➢ Understanding infinite behavior: With the help of formal tools, the power of mathematical reasoning allows us to ask and answer questions about the behavior of models in an infinite period of time. For example, we can ask whether the model can be stuck forever without reaching the desired state, or whether it can be guaranteed to eventually perform certain behaviors that cannot be guaranteed within a limited time limit. These problems cannot be effectively solved by techniques such as simulation.

2.2 Category of formal verification

2.2.1 Theorem proving

Theorem proving is the process of using mathematical reasoning to verify whether the implemented system meets the design requirements (or specifications). It is a method of formal verification based on evidence, as shown in Fig 1.

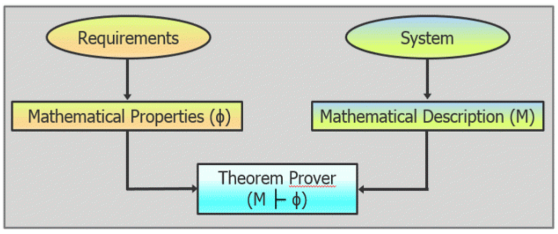

Fig 1: Theorem proving

The biggest advantage of the theorem proving is that it can handle very complex systems. However, the proof of the theorem is not completely automatic and requires manual intervention to complete, which requires time and the professional knowledge of the operator. Moreover, in the case of a failed proof, no counterexamples will be generated, which also makes it relatively difficult to locate errors.

2.2.2 Model checking

Model checking, also called property checking, is a state-based formal verification method, as shown in Fig 2.

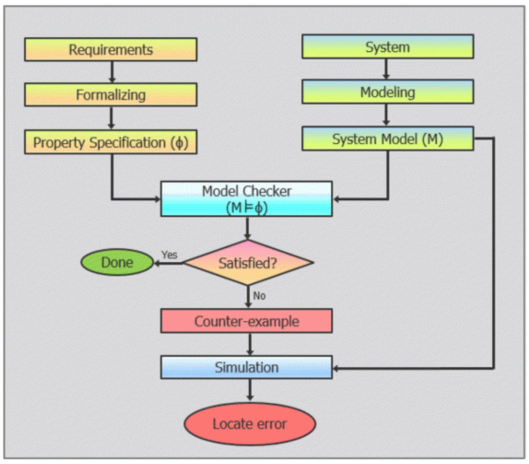

Fig 2: Model checking

Once the system model and property specifications are provided to the model checker, the verification process will be completely automatic. However, judging from the number of states to be processed by the model checker, there is a problem of state space explosion when applied to large systems.

2.2.3 Equivalence checking

Equivalence checking is the process of verifying whether two designs are functionally identical, which is divided into logical equivalence check and sequential equivalence check:

➢ Logical Equivalence Check (LEC): Also called Combinational Equivalence Check. It is the process of verifying that the two designs have the same combinational logic between the registers, as shown in Fig 3. The two designs being compared should also have the same number of registers. This technique is used to verify whether two designs at different levels of abstraction are functionally identical; for example, whether an RTL design is functionally the same as a gate-level netlist.

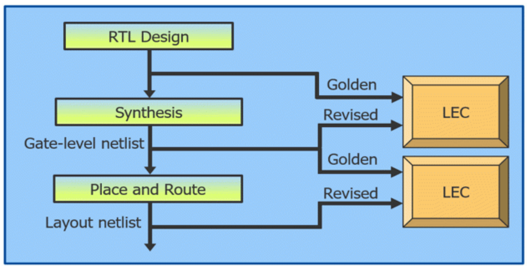

Fig 3: Logical equivalence check

➢ Sequential Equivalence Check (SEC): Sequential equivalence check is the process of verifying whether two designs are the same in function, and verifying whether there is the same output when the same input is provided, as shown in Fig 4. It is used to compare the sequential logic of two designs, and the two designs may have different implementations, such as checking to compare the modified design with the standard design, and verifying whether they are functionally consistent.

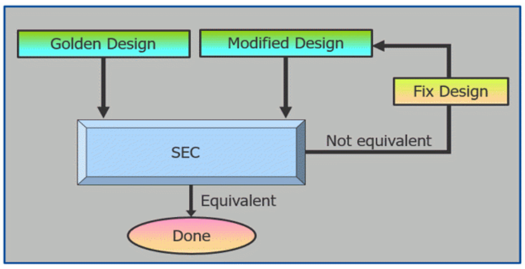

Fig 4: Sequential equivalence check

2.3 Formal verification of ARM-V8 cores

In the formal verification of processors, most of the work is focused on verification with high-level model of the micro-architectures. For example, Lahiri et al. [3] verified the micro-architecture of the M* core (early RISC architecture) [4]. Through a series of continuous improvements, the micro-architecture of the out-of-order processor has been verified. But real design of the RTL model has not been verified. The challenge of RTL-level formal verification is that it is difficult to use abstract technology, because the optimization process of high performance processors makes it difficult for actual RTL to separate sub-modules that directly match the original specification.

ARM released the ARM-v8.2 machine-readable specification in 2017, which was written in ASL (ARM Spec Language). Before released officially, ARM has established a formal verification process according to the spec and it's former version written in pseudo codes. The formal verification flow has run on multiple projects. In the formal verification of the ARM V8 core, Reid et al. chose RTL-level verification instead of verifying the high-level model of the micro-architecture design according to the specification, in order to avoid introducing new errors in the process of converting the high-level micro-architecture model into RTL. In [2], Reid gave a detailed description of the formal verification process for instructions built on the ARM-v8A/M cores.

2.3.1 Method of ISA-Formal

ISA-Formal is an end-to-end verification VIP based on the model checking method, and directly verifies the entire path from instruction decoding to instruction retire according to the ARM architecture specification. The idea of this formal verification is as follows:

Suppose the core starts from the microarchitecture state uArch0, there are no instructions in the pipeline, and then executes multiple cycles, each of which may issue an instruction. This is used to put the processor in a complex state where risks, forwarding, etc. may occur. Finally, execute an instruction In and test whether the instruction is executed correctly. Before the execution of In and after the execution of In, the abstract function abs is used to abstract the microarchitecture state into the architecture state. The pipeline is not flushed before or after In, as shown in Fig 5. Modeling and checking carried out on each single instruction. This can effectively reduce the number of state spaces, and make formal verification applicable at the RTL level.

$$\begin{tikzcd}
uArch_0 \arrow[r, "I_1"] & uArch_1 \arrow[r, dashed] & uArch_{n-1} \arrow[r, "I_n"] \arrow[d, "abs"] & uArch_n \arrow[d, "abs"] \\
 & & Arch_{n-1} \arrow[r, "I_n"] & Arch_n
\end{tikzcd}$$

Fig 5: Abstract microarchitecture state of processor to architecture state

Taking a 5-stage sequential processor as an example, the addition instruction can be modeled as shown in Fig 6. When the ADD instruction commits, the micro-architectural register file has the same values as the architectural register file before ADD instruction executes. At the end of the execution, the values of the micro-architectural register file and the architectural register file contains the results of the execution of the ADD instruction. Therefore, the before state can be obtained by reading the state at the end of the write-back phase, and the state after can be obtained from the end of the MEM phase.

Another important input required for property checking of instruction execution is the opcode of the current instruction. Opcodes are usually discarded shortly after the instruction is decoded, and are not available when the instruction is issued. Therefore, it is necessary to implement an helper-logic to copy the opcode from one stage to the next, and implement the same pipeline stall/flush logic as the data path. The follower and abstract logic of the pre/post state are shown in Fig 6.

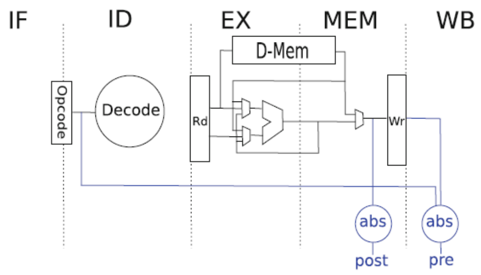

Fig 6: Modeling of ADD instruction

Generally, spec of any individual instruction can be written as a short piece of purely combinational logic.

For example, look at an ARM's 16bit add instruction ADD Rd, Rn, Rm. The opcode is 0b0001100 | Rm << 6 | Rn << 3 |Rd, which add up contents of Rn and Rm, and write to Rd.

In Systemverilog, it is as follows:
```
assign ADD_retiring = (pre.opcode & 16'b1111_1110_0000_0000) == 16'b0001_1000_0000_0000;
assign ADD_result = pre.R [pre.opcode[8:6]] + pre.R [pre.opcode[5:3]];
assign ADD_Rd = pre.opcode [2:0];
```
And assertion to check its execution is:
```
assert property (@(posedge clk) disable iff (˜reset_n)
ADD_retiring |-> (ADD_result == post.R [ADD_Rd]));
```

Now the property check for 16bit ADD instruction is established.

2.3.2 ARM-V8 ISA formal spec

Use tools to process ASL based machine-readable architecture documents, and extract the encoding diagram of each instruction, the ASL code used to decode the instruction, the ASL code used to execute the instruction, and any supporting code needed to execute the instruction, as well as look up the decode tree of the instruction corresponding to the given positioning mode. This also includes the ASL code of the system architecture: page table traversal, exceptions, debugging, etc. Then convert it into a SystemVerilog model to support a commercial formal verification tool based on model checking.

The errors that can be captured using the above methods include: decoding errors, data path errors, and interaction errors between instructions. Especially the interaction between instructions, because of the diversification of the micro-architecture, the control logic is prone to errors, and the errors are difficult to find through conventional simulation based tests, while formal methods can cover them all.

III. Application of Formal Verification in RISC-V Core Design Verification

3.1 Challenge in RISC-V core verification

As mentioned before, in order to adapt to a wider range of applications, the RISC-V instruction set definition is flexible and extendable, which makes it particularly difficult to verify in three aspects.

Firstly, the RISC-V instruction set has many optional functions and possible changes [5,6]. The 32 registers of the processor can be 32-bit, 64-bit or 128-bit. The Baseline "I" instruction set has an optional "E" version, which only supports 16 32-bit registers for embedded applications. The instruction set defines three privilege levels, nine exceptions related to privileged instructions, and 4096 control and status registers. Rich options such as "M/A/F/D/Q/C" increase the complexity of verification. At the same time, the instruction set allows the definition of custom instructions, and these instructions must be verified to ensure that they work properly without violating standard instructions.

Secondly, the RISC-V ISA is designed to map to many different micro-architectures, from micro controllers to multi-core implementations with the most advanced processor functions. Different micro-architectures must comply with the functions specified by the architecture to ensure correct execution of instructions.

Finally, any RTL-level design must be statically analyzed to eliminate common coding errors.

Any method based on simulation or emulation can only explore part of the design function and cannot ensure that constrained random testing or manual testing finds all hardware errors; only formal verification can provide this guarantee. Similarly, only formal verification can prove that the design does not happen that shouldn't happen. In this case, certain parts of the design may pose risks to the final application with high security or trust requirements.

3.2 Formal verification on design exercise

When the design is only partially completed, we use the formal verification tool to quickly explore and analyze the RTL model, which can effectively find some errors in the design. For example, Synopsys VC-Formal's AEP and FCA app, Cadence JasperGold's SuperLint app, can automatically complete array boundary checking, mathematical calculation overflow (Arithmetic Overflow), X assignment, set/reset cannot occur at the same time, conditional execution (full case/parallel case)), multi-drive, deadlock and state machine check. In addition, the tool can also analyze code and state reachability, jumps between states, etc. These checks do not need writing assertions manually.

In addition, a preliminary check of the most basic functions of the design can be made by writing some simple assertions. These lightweight property checks can start the verification without establishing a testbench before the design is completed, and find bugs as soon as possible.

3.3 Formal verification on bug hunting mode

The Bug Hunting mode of FPV (Formal Property Verification) is a medium-effort formal verification, which is usually applied to modules with a certain degree of complexity. The module also needs to use simulation or emulation and other supplementary verification methods. The purpose of using the Bug Hunting method is to find corner cases that are unlikely to be hit by simulation.

When a certain part of the design is considered to be particularly risky due to a large number of corner cases, and even after running a complete random simulation verification, a bug hunting FPV can be used to have more confidence in the verification results.

When planning a bug hunting FPV, constraints can be used as needed to simplify the model being processed in order to focus on the current problem domain.

3.4 Formal verification on full proof

Full proof FPV is the traditional goal of formal verification research, usually for complex or risky designs, aiming to fully prove that the design meets its specifications. If done well, full proof FPV can be used to completely replace the unit-level simulation test.

At the same time full proof FPV needs the most effort, because it involves trying to prove the complete functionality of the design. The design that has fully passed the FPV proof, has a set of reliable assertions with complete definitions and specifications, and a set of reliable and fully reviewed coverage points and hypotheses. Compared with random regression in a complete simulation environment for several months, it provides a higher degree of confidence. This method also reduces the risk of hard-to-find errors and nightmares due to corner cases that were not anticipated during the test plan, such as Intel CPU floating-point calculation errors and Spectre and Meltdown vulnerabilities.

3.5 RISCV-formal

The open source project riscv-formal [7] on github is an end-to-end formal verification VIP for the RISC-V core. Similar to the method of ARM ISA-Formal, riscv-formal establishes a SystemVerilog model for each instruction of the RISC-V instruction set, and can realize the BMC (bounded model check) formal verification of Verilog design with commercial model checking tools.

Riscv — formal has realized two types of formal verification:

- Instruction Formal Verification

Prove that all completed instructions match the state transitions of the register file or flags before/after the instruction is executed.

- Consistency Formal Verification

Prove that the sequence of state transitions is consistent, including:

Register write followed by register read must read back the previously written value;

The pc value of two consecutive instructions must match;

The reordering of instructions must be causal;

It is impossible to read the value from the register before writing.

We established a formal verification environment based on riscv-formal on the VC-Formal platform, and performed formal verification on the open source core PICORV32. Under the condition of setting the verification depth to 20, we completed the BMC verification of instruction set I and C. When part of the actual core design is completed, a formal verification platform can be established on this basis, and a more powerful verification can be performed by adding custom instruction models, etc.

IV. Conclusion

While facing complex scenarios such as processor verification, traditional simulation verification based on UVM cannot guarantee coverage of corner cases due to its own limitations. The defects in verifying the correctness and safety of the design cannot be ignored. With the enhancement of server computing power and the optimization of formal algorithms, formal verification has become a new choice in the design verification of RISC-V cores. BMC for the purpose of bug hunting as a supplement to simulation verification can ensure the complete inspection of specific functions, and the establishment of a complete formal specification of the architecture document can achieve complete proof of design.

References

[1] Sligman E., et. al, Formal Verification: An Essential Toolkit for Modern VLSI Design. Elsevier, 2015.

[2] Reid A., et.al, End-to-End Verification of ARM Processors with ISA-Formal. Proceedings of the 2016 International Conference on Computer Aided Verification (CAV'16), S. Chaudhuri and A. Farzan (Eds.). CAV 2016, Part II, Lecture Notes in Computer Science 9780 (July 2016), 42–58.

[3] Lahiri, S.K., et. al, Experience with term level modeling and verification of the M*CORETM microprocessor core. In: Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop 2001, Monterey, California, USA, 7–9 November 2001, 2001, 109–114

[4] Lahiri, S.K., et. al, Deductive verification of advanced out-of-order microprocessors. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, 2003, 2725, 341–354.

[5] The RISC-V Instruction Set Manual. Volume I: User-Level ISA, Document Version 20191213.

[6] The RISC-V Instruction Set Manual. Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified.

[7] https://github.com/SymbioticEDA/riscv-formal