Wedging with formal verification to strengthen the quality of custom RISC-V SoC

Nicolae Tusinschi[1], Salaheddin Hetalani[1], Pascal Gouedo[2]

[1]Siemens EDA
[2]Dolphin Design

Abstract

On the path of creating high-quality processor IP a major challenge is the need to verify the correctness of the design across a wide range of configurations and use cases. Each configuration needs to be validated and can be challenging to ensure all possible input patterns are exercised and functions are correctly implemented. In this paper, we go through a novel approach in verifying an open-source, high-quality and customizable RISC-V core with a comprehensive methodology. We explain the importance of thorough planning, use of automated code review and exhaustive formal guided automation. We provide evidence illustrating how proposed methodology is cost-effective and highly efficient in preventing bugs to escape and accelerates sign-off process.

Introduction

Freedom, how could we describe it better in just one word? Joy, peace, autonomy, or maybe choice? We all know from early age freedom is important. It is one of most fundament human rights to which everyone is entitled. Freedom allows for self-expression, allows for individual autonomy, promotes creativity, and encourages change by removing barriers, challenging existing norms and creating new opportunities.

RISC-V is one example of how unprecedented freedom enables adoption, enables creativity, enables performance. It is a highly customizable instruction set architecture (ISA) with many standardized extensions and reserved space for custom extensions. It is supported by a mature software toolchain from compiler to full Linux running on RISC-V. Many micro-architectures available covering full range of applications from simple embedded cores to deep multi-threaded out-of-order implementations.

CV32E40Pv2 is an embedded class open-source core maintained by OpenHW Group [1]. It is a four-stage single-issue in-order implementation. Highly configurable supporting, in addition to I, M, C extensions, floating point extensions F and Zfinx, and the custom extensions Xpulp and Xcluster, i.e., post-incrementing load and store, hardware loops, etc. The core is a key component for IP platforms for energy efficient SoC, be them MCU subsystem or multicore DSP and requires meeting high quality standard.

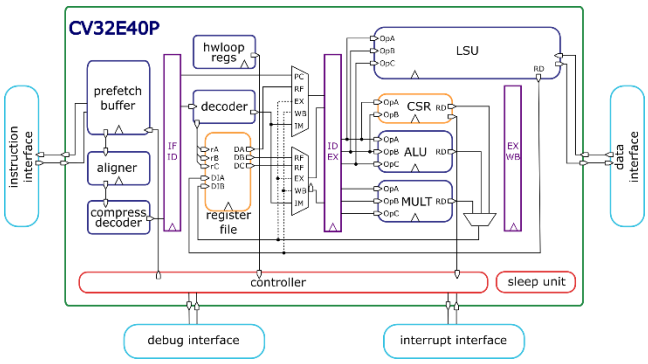

Figure 1: OpenHW CV32E40Pv2

Next, we propose a formal methodology we deployed in verifying CV32E40Pv2 alongside simulation significantly reducing the effort and delivering exhaustive, complete verification.

Processor Verification Methodology

The following sections describe in detail the processor verification methodology.

1. Verification planning

Before any verification is started there is a need for thorough planning. As part of the proposed methodology, we have included the planning phase as a mandatory step. Detailed formal verification plan is generated automatically by the tool for each of the implemented extensions, as described in section 5, introduced later. The verification plan specifies how an instruction is decoded based on the RISC-V ISA specification [2], and how it's supposed to be executed based on the Sail language, the formal specification language adapted by the RISC-V International Organization [3]. In addition, each implemented instruction has a test plan entry describing the verification method applied with annotated result to it.

Sub Feature	Disassembly	Decoding	Sail Description	Assertion
ADDI	addi {rd},{rs1},{imm}	imm[11:0] rs1 000 rd 0010011	X(rd) = X(rs1) + EXTS(imm)	RV32I.ARITH_a
LUI	lui {rd},{0ximm}	imm[31:12] rd 0110111	X(rd) = EXTS(imm)	RV32I.ARITH_a
AUIPC	auipc {rd},{0ximm}	imm[31:12] rd 0010111	X(rd) = PC+EXTS(imm)	RV32I.ARITH_a
ADD	add {rd},{rs1},{rs2}	0000000 rs2 rs1 000 rd 0110011	X(rd) = X(rs1) + X(rs2)	RV32I.ARITH_a

Figure 2: Snapshot of the generated verification plan

2. *Verification environment setup*

Once the design is read-in, applying RISC-V processor verification app to RTL is straightforward and requires constraining memory interfaces before going through the app flow itself, explained in detail in the following sections. For this purpose, bus protocol compliance verification libraries can be used to constrain the memory interfaces behavior allowing only legal input sequences.

3. *Automated design analysis*

The first step of the processor verification app flow is the automated design analysis that is conducted on the core RTL implementation, where architecture and microarchitecture details are extracted automatically and stored in a JSON database. Examples of extracted information includes RISC-V extensions, privilege levels, width of the integer register, number of counters implemented, and all details related to standard CSRs on bit level, whether they are implemented and what are the corresponding RTL signals representing them.

4. *Adding core specific information*

Details that are specific to each core, such as the RTL signals representing the memory interfaces, or RISC-V parameters that can be overwritten depending on whether a certain feature is implemented or not, have to be provided, such that the database is updated. This can be done in two different ways: either using the app's GUI or in a scripted way using a JSON file that can be merged into the initial database extracted.

If custom extensions are present, users have to provide their details in a similar manner. Adding a custom instruction is a matter of specifying how it's decoded and how it's supposed to be executed. Similarly, custom registers, CSRs, or register files can be specified. Noting that if custom CSRs are implemented, users have to provide their information before going through the automated design analysis. This way, the extraction is aware of their addresses and can map them to the actual RTL signals.

5. *Automated assertion generation*

Once the database is updated, the verification plan and all the verification files, written in SVA, including assertions applicable to the core are automatically generated. One of the generated files, for instance, is a bind file mapping the core RTL design signals to their counterparts of the assertion template file, containing an already defined set of properties.

6. *Retuning assertion template*

In this step of the flow users might be required to refine the initial assertion template, in terms of specifying the state when an instruction is ready to be executed, and properties' timing details based on the pipeline microarchitecture.

7. *Running assertions and pinpointing bugs*

Assertions are run to get either counterexamples that root cause corner case RTL issues, or full proofs indicating that the RTL satisfies the assertions with no added or undocumented functionality being implemented.

Application Results

Out of all the possible 16 different configurations the CV32E40Pv2 core could be set to, 5 were of interest to be verified. Each configuration enables different set of RISC-V extensions. On average around 400 assertions were generated per configuration, considering that the Xpulp extension, enabled in all configurations chosen, adds around 320 instructions to be verified, where one assertion at least is needed for each of the custom instructions.

Applying the proposed processor verification methodology resulted in identifying 31 issues. These are classified as illustrated in Table 1 and their detailed descriptions can be found in the core's GitHub repository page [4]. Convergence wise 100% unbounded proofs were achieved, considering that for multiplication and division instructions the verification is partial, i.e., only partial bits of operands were allowed to toggle.

Table 1: Issue classification

# Issues	Issue Type
1	Fetch memory access
2	Data memory access
6	Illegal instruction exception
8	Register and CSR update
12	Xpulp specification
2	OBI protocol violation

Summary

The key component in successful verification is to match the right methodology to the right problem. With proposed methodology we have accelerated verification speed up by a factor of ten with faster coverage closure, optimized formal engines and pinpoint debugging.

References

[1] https://github.com/openhwgroup/cv32e40p
[2] https://riscv.org/technical/specifications/
[3] https://github.com/riscv/sail-riscv
[4] https://github.com/openhwgroup/cv32e40p/issues