How the Right Mindset Increases Quality in RISC-V Verification

Philippe Luc, Codasip, 2474 Route Départementale 6007, 06270 Villeneuve-Loubet, France – philippe.luc@codasip.com

Salaheddin Hetalani, Siemens EDA, Nymphenburgerstraße 20a, 80335, Munich, Germany – salaheddin.hetalani@siemens.com

Nicolae Tusinschi, Siemens EDA, Nymphenburgerstraße 20a, 80335, Munich, Germany – nicolae.tusinschi@siemens.com

***Abstract*-The RISC-V open standard has generated increasing interest across almost all applications. This appetite for more freedom in processor design is shifting the verification responsibility to a growing community of developers. Processor verification, however, is never easy. Having the right mindset can make a huge difference in the quality of your processor. Specifically, with processor verification it is critical to leverage the strengths of multiple forms of verification. In this presentation, we will focus on the contribution that automated formal verification can make in a larger "swiss cheese model" adapted from the avionics world to improve the quality of RISC-V processors by "breaking the design" and testing to the edges in order to find more bugs.**

I. Introduction

RISC-V gives the ability to customize and/or extend the ISA for a given application through architectural and microarchitectural modifications, to meet end-users' unique requirements. This design freedom, however, may bring new challenges. The very novelty and flexibility of the new specifications - and correspondingly new logic - can inadvertently create specification and design bugs that are difficult to find. Verification is a crucial yet costly part of any design project, so it is of paramount importance to find the best possible solutions that will ease RISC-V adoption and meet the high-quality standards customers expect.

II. The right Verification Approach

One very important aspect of developing high-quality processor IP is to have the right mindset for verification. To achieve best-in-class verification, the verification team's mindset can make the difference between creating a verification plan that will catch bugs vs. a collection of tests that make a nice-looking coverage report but will fail to find deep corner case bugs.

A typical bug curve says finding around 1,000 bugs in a processor design process is expected; and naturally the number of bugs increase if we think about cores with multi-issue and out-of-order execution pipeline capability. Typically, these bugs can be classified as we see in Fig.1, into easy, medium, hard, and late bugs on time axis.

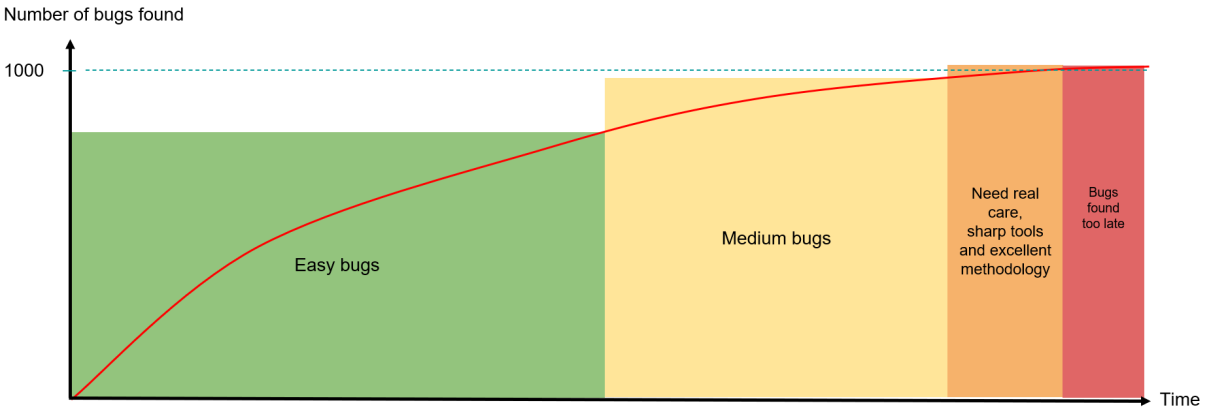

Fig. 1: Processor design bug curve

The majority of the bugs are “easy” – then come the “medium” bugs. There are only a few bugs that are difficult to identify, and the ones found too late in the project are critical.

What do we classify as easy bugs? Generally, bugs that are identified simply by running a test exercising the feature. In addition, we have bugs that are simple errors introduced by designers by missing a semicolon or duplicating a condition. Automated formal checks greatly help clean the design as they help eliminate the easy bugs and allow the team to focus on the issues that are really hard to catch. In the case of specification features that are not yet implemented, for example, performing a code review with the specification at hand helps quickly identify inconsistencies.

Medium and hard to identify bugs are more complex and require a targeted approach – starting with good checkers. Quite often medium and hard bugs are triggered by asynchronous events or even unthinkable combinations for engineer’s judgment. But what puts these bugs into one bucket is that we generally know where to search, and we target them with a full suite of constrained-random simulation testing, with coverage collection enabled.

Now what about late bugs? Ultimately what plays a huge role in identifying these is the approach engineers are taking – the strategy on how you go about finding your next bug. Specifically, how to learn from previous bugs to catch the new ones. Are you looking for new bugs by looking to improve your current constrained-random testbench; or are you looking in the places where you don’t have a testbench and must write new tests? It’s at this stage where the right verification approach is the key.

III. Swiss Cheese Model Verification methodology

The Swiss cheese model is not new and has its roots in the avionics industry [1] and there is a great parallel to how processor verification should be approached. The important bit is that if there is a bug in the field (e.g. the late bugs in the red zone in Figure 1 above), then there is a direct path through all cheese layers from final deliverable to RTL. In order to avoid a direct path through the whole block, at least one cheese layer must interrupt the path from one end to the other. In the aviation industry, there are several layers to ensure safety. For speed sensors (pitot tubes), they should be visually controlled during the pre-flight visit, checked for valid measures just before take-off; and there might be redundant components added, and added accessories like heaters in case of frost, etc. The point is: all of these measures ensure that a potential failure is prevented or detected before an accident happens. To reach high quality in RISC-V designs, we must adopt a similar mindset where no single verification method is enough on its own – we must apply different verification methods to continuously improve on coverage, quality of code reviews, OS boot flows, and more.

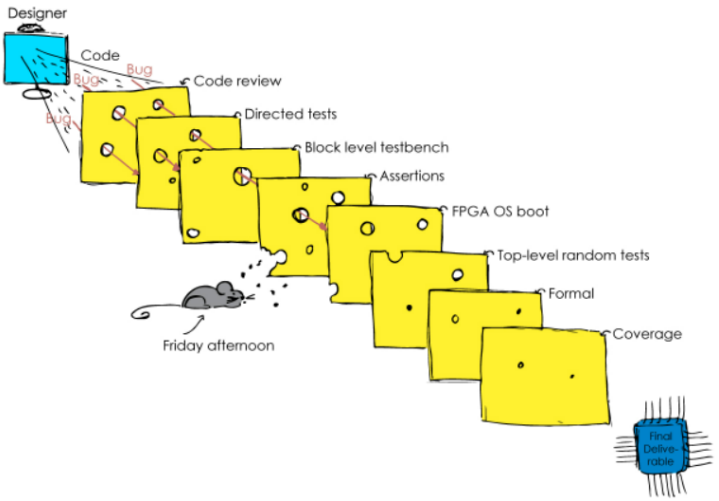

Fig. 2: Swiss Cheese Model Verification methodology [2]

Processor verification usually uses the methodologies from Fig. 2.
It combines different approaches:

- Static code analysis, using best-in-class software development usages, such as code review with merge request, and lint checks.
- Simulation-based testing, with variations on test topologies, checking mechanisms, and coverage.
- FPGA-based testing, booting rich OSes, and stress tests.
- Formal-based testing.

By using different slices of cheese or verification methods we have the following benefits:

- Redundancy ensures continuity if one layer fails.
- When bugs are found in development, it indicates that there were holes in several slices. Improving the verification methods reduces the size of the holes in each slice. We improve our chances of detecting bugs, from easy bugs to corner cases, and from simple to complex bugs.
- The potential of each verification technique is maximized.

The greater the number of holes and the bigger the holes, the bigger the chance of bug escaping detection. If the same area of the design (cheese slice) is not covered by any of the verification techniques (overlapping holes between the slices), a bug will make it through the slices undetected and may end up in the final deliverables. A good verification methodology will present as few holes as possible, and those holes will be as small as possible on each slice. In short, a single verification technique cannot do everything by itself, it is the combined action of all of them that improves the overall quality of the verification and hence the processor quality.

On standard projects, most of the engineering effort is spent on simulation-based testing. A common approach is to have a random test generator (being top level or block level), and simulation applying this test to the Design Under Test (DUT), with various checkers (scoreboard, assertions, comparaisons with model). While these methodologies have successfully delivered quality chips, there is room for improvement. Simulation based testing usually needs several thousands of tests running in parallel for months to reach saturation of the random generation capabilities.

Building on this is – adding a key slide to the verification stack – is the addition of Formal analysis technology Leveraging an exhaustive analysis under-the-hood, and reflecting the DUTs specification in formal terms – via SystemVerilog assertions, cover properties, and constraints – you make the cheese holes smaller and are more likely to find unforeseen design paths hiding bugs that you would not find with the other slices/techniques. This enables engineers to find bugs of high importance earlier in the development process. And with the added mindset of continuous improvement, based on the bugs reported by formal all the other slices can be improved as well. This brings better simulation-based testing, leading to some other collateral bugs found.

IV. AUTOMATED RISC-V FORMAL VERIFICATION APP

The addition of formal analysis comes via a formal-based, easy-to-deploy RISC-V processor verification app that's designed to target late bugs by bringing a high degree of automation with a dedicated set of assertion templates for each instruction. This automated, exhaustive flow is a reflection of the very cautious verification mindset discussed above. Specifically, this approach leverages an extension of IEEE standard SystemVerilog Assertions (SVA) that provides high-level, non-overlapping assertions that capture end-to-end transactions and requirements in a concise, elegant way. The benefits of this methodology include:

- The ability to translate functional requirements for automatic detection of specification omissions and errors, holes in the verification plan, and unverified RTL functions
- The capturing of entire circuit transactions in a concise and elegant way, similar to timing diagrams
- Clean separation of implementation-specific supporting verification code from reusable specification-level code
- Achieving 100% functional coverage with high-level and easy-to-review assertions

In the next sections, we will show how this flow captures the high-level operational view of the DUT, and how it maps to sequential or pipelined implementation, out-of-order execution, and other possible options in the RTL core. This framework also splits the specification side from mapping to implementation to enable full SVA reuse.

The RISC-V processor verification app models the RISC-V ISA specification [3] in SystemVerilog (SV) and uses the unique OneSpin's 360 operational assertion modeling principle to create assertions verifying the whole core RTL implementation. Based on this modeling principle, the overall design specification is broken down into a list of operations, which forms a complete specification. An operation is a multi-cycle activity of the DUT, e.g. instruction execution in a processor, as shown in Fig. 3. These operations are captured formally by the so-called "operation properties", expressed via the Timing Diagram Assertion Library (TiDAL) that extends SVA with constructs that capture timing behaviors more intuitively.

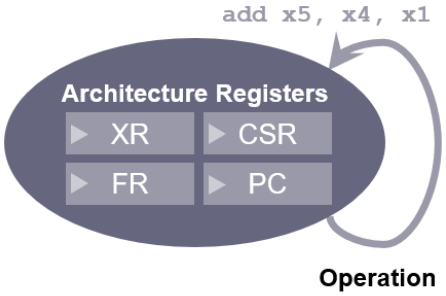

Fig. 3: Operation example

An operation property, illustrated in Fig. 4, describes an implication and is characterized by two main parts:

- Trigger part describing the cause, which consists of conceptual start state, and input and architecture register triggers
- Monitor part describing the effect, which consist of conceptual end states, and expected outputs and architecture register updates

In other words, an operation property/assertion verifies the intended behavior of an operation whenever it's triggered by some input and architecture register values, by checking the actual output values and new values of architecture registers. The new values of architecture registers become current values for the next operation and so on. What is meant by architecture registers are the integer and floating point registers, control and status registers (CSRs), and the program counter (PC). If present, custom registers or register files are also part of the architecture registers.

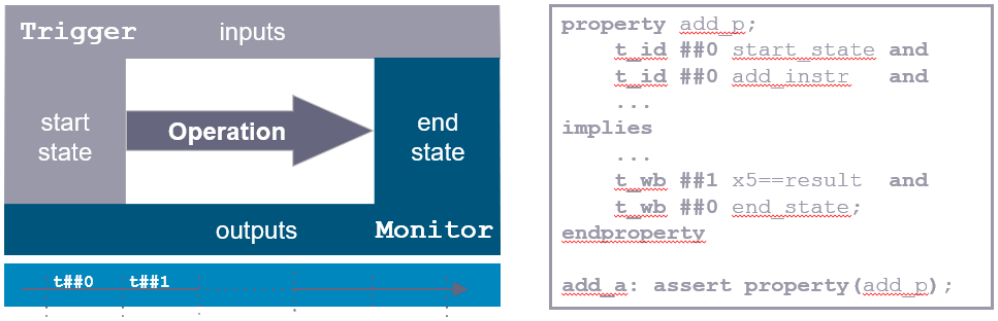

Fig. 4: Operation property example

Once the design is read-in, applying the RISC-V processor verification app on the core RTL is straightforward and requires only constraining memory interfaces before going through the app flow itself, explained in detail in the following sections. Bus Protocol Compliance Verification IP apps can be used to constrain memory interfaces behavior, such that only legal input sequences are allowed, depending on the bus protocol used. Prior to starting this verification process, however, we recommend using the state-of-the-art, automated formal-based Auto-Checks, to sanitize the code in early development stages since a cleaner design means less effort for later functional verification.

The so-called Auto-Checks is a set of synthesized assertions, generated automatically by a self-acting inspection of the design, that are executed by formal engines and verify whether RTL designs satisfy certain behavioral quality criteria. These criteria include, exclusion of predefined illegal conditions (out-of-bounds array accesses, occurrences of 'X' values, division by zero, etc.), detection of unreachable code and constant signals, correct initialization of signals, and observation of user-defined inline assertions. Successful checks guarantee that illegal conditions are excluded in every possible execution trace of the design. Violations are reported along with counterexamples, which show how the critical situations arise.

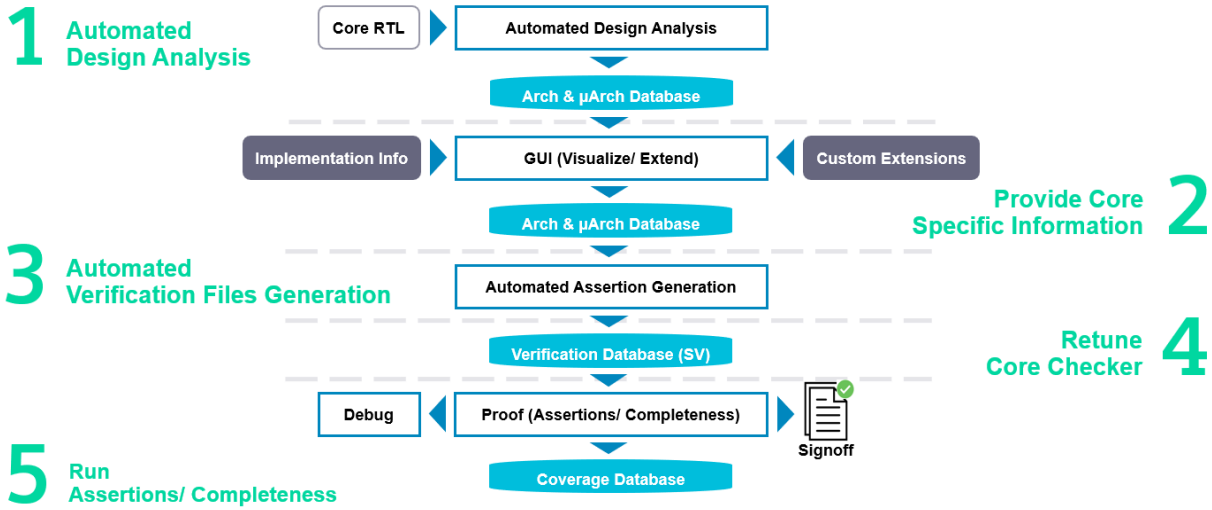

Fig. 5: RISC-V processor verification app flow

V. Automated Design Analysis

As indicated in Fig. 5, the first step of the RISC-V processor verification app flow is the "Automated Design Analysis" that is conducted on the core RTL implementation, where architecture and microarchitecture details are extracted and stored in a JSON database, for later usage. Examples of architecture information include, RISC-V extensions, privilege levels, width of the integer register, and number of counters implemented. Examples of micro-architecture information include, all details related to standard CSRs, whether they are implemented and what are the corresponding RTL signals representing them, on bit level. All data stored in the database is populated into the app's GUI, as shown on Fig. 6, where users can navigate through them.

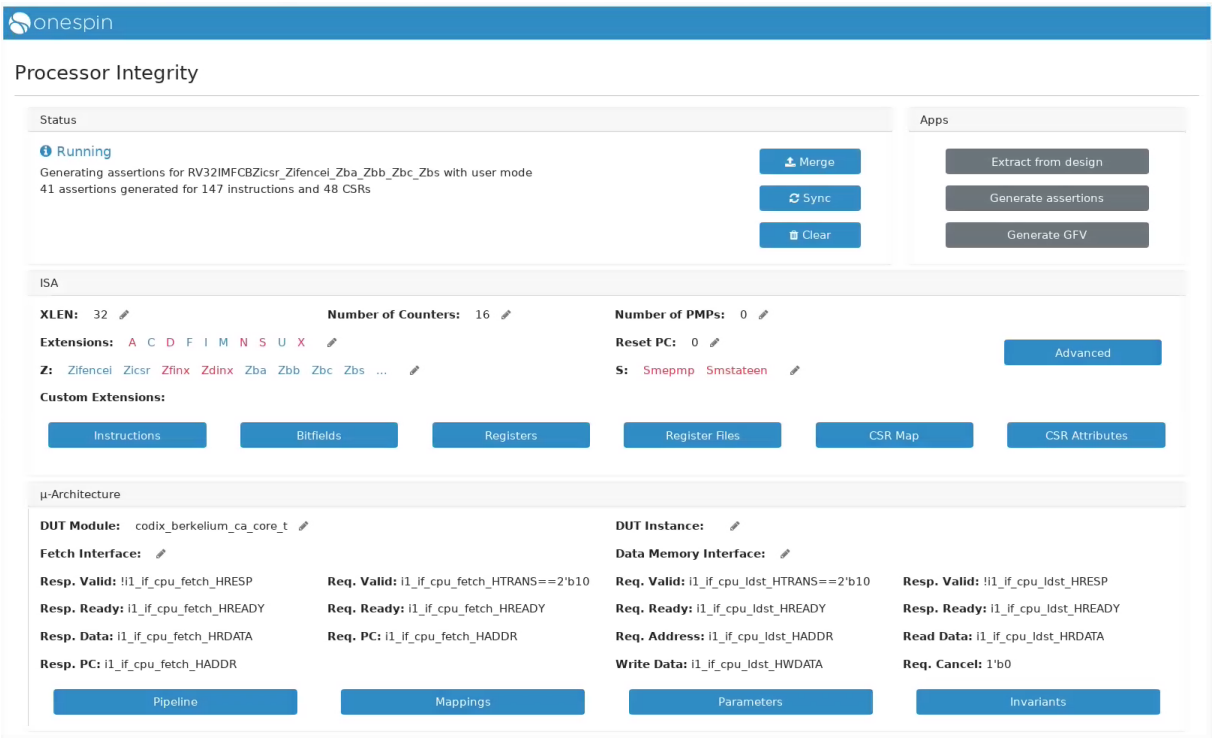

Fig. 6: RISC-V processor verification app GUI

VI. Provide core specific information

Here comes the second step of the flow, “Provide 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 RISC-V feature is implemented or not, have to be provided. This can be done in two different ways: either using the GUI or in a scripted way using a JSON file that can be merged into the initial database extracted.

VII. Generate verification files

Once the database is updated, we are ready to generate the verification files and all assertions applicable to the core, written in SVA. This “Automated Assertion Generation” is the third step of the flow. 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, the “core checker” containing an already defined set of TiDAL properties, introduced earlier. Furthermore, a TCL file containing disassembly information for all instructions applicable to the core is generated, which is used to assist users knowing what kind of instruction is currently being executed in the pipeline while debugging counterexamples for instance.

VIII. Retune core checker

The next step of the flow, “Retuning the Core Checker”, might require users to refine the initial property template, in terms of specifying the conceptual state, that is, when the instruction is ready to be executed, and time related information.

IX. Verify properties

In the last step, users are ready to “Run Assertions” and get either counterexamples that pinpoint corner case RTL issues, or full proofs indicating that the RTL satisfies the assertions with no added or undocumented functionality being implemented.

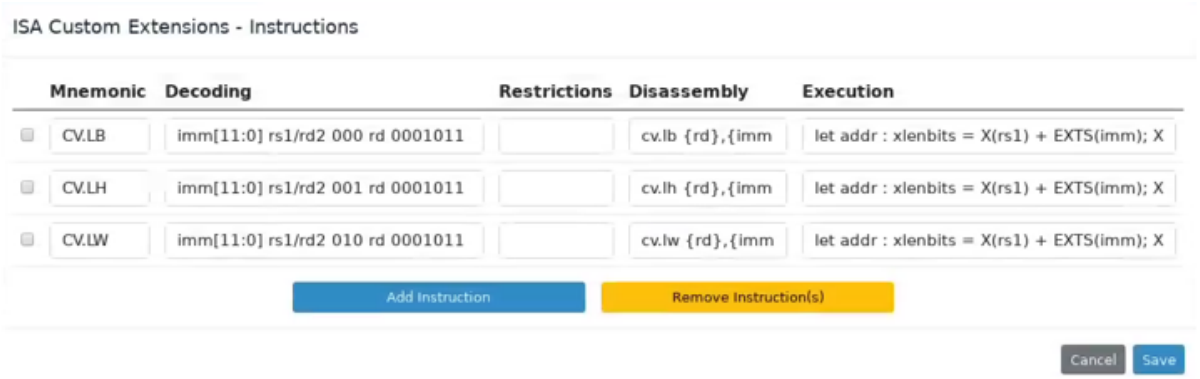

Fig. 7: Custom instructions modeling

If custom extensions are implemented, users can specify their details in the second step of the flow. As illustrated in Fig. 7, adding a custom instruction is a matter of specifying how it's decoded based on the RISC-V ISA specification, and how it's supposed to be executed, which is specified using the Sail language, the formal specification language of the RISC-V architecture [4] adapted by the RISC-V International Organization [5]. Similarly, custom registers, CSRs, or register files can be specified. It is worth noting that if custom CSRs are implemented, users have to provide their information to the app before going through the first step of Automated Design Analysis, using the GUI or by merging a JSON file containing their details as mentioned earlier. This way, the extraction is aware of their addresses and can map them to the actual RTL signals.

X. Results

We will use the small and energy-efficient Codasip low-power embedded core, L31, as an application example of the verification methodology introduced, with focus on the results of formal verification applied. L31, shown in Fig. 8, implements RV32IMFCBZicsr_Zifencei_Zba_Zbb_Zbc_Zbs with machine and user privilege level modes and external debug support. It has a single 3-stage in-order execution processor pipeline and AHB-Lite memory interfaces.

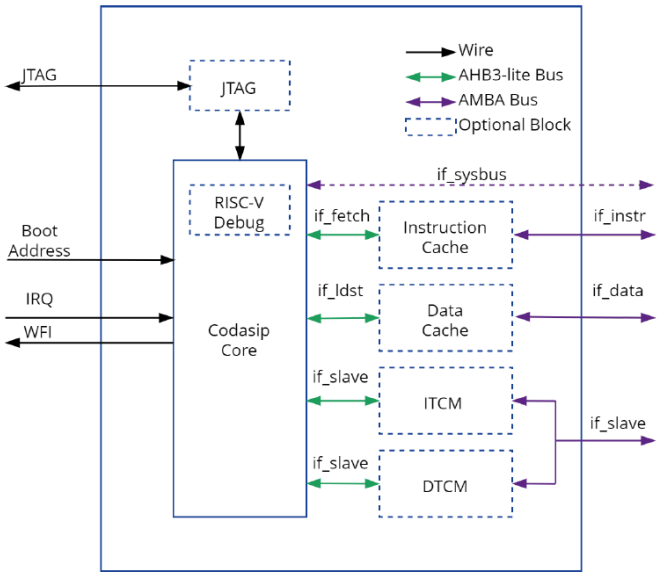

Fig. 8: L31 Codasip RISC-V processor block diagram

Starting with the Auto-Checks application, we were able to identify some late bugs that are "easy". E.g., a very simple bug that formal automated check identified was a tied to zero clear flag which caused an RTL branch to not be executable, illustrated in Fig. 9.

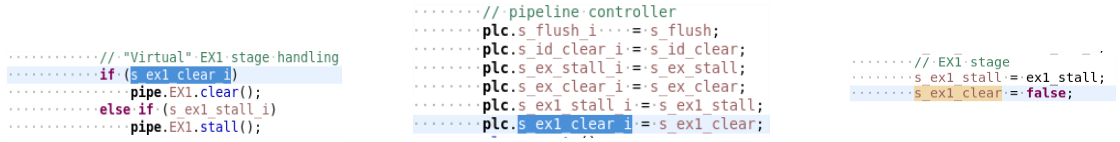

Fig. 9: Auto checks and dead code analysis found a signal that was hardwired

We constrained the AHB-Lite fetch and data memory interfaces by instantiating the associated bus protocol compliance verification IPs. Then we applied the RISC-V processor verification app to get the set of assertions applicable to the core. Bug hunting was our first focus; thus, we first verified the basic I, C, Zicsr, and Zifencei extensions by running the associated assertions after refining their property timing and the unified conceptual state. A total of 12 bugs were identified after only 28 hours of application, a time that we couldn't achieve in our simulation-only flow, not to mention the quality of bugs found. Targeting the other extensions like, M, F, and B resulted in catching 10 more bugs. Achieving shorter assertion run times was our next target, for which black-boxing techniques on signal level, like cutting out counters, were applied. The bugs found included:

- Illegal instruction exceptions not raised
- Illegal CSR counter increment
- Legal instructions treated as illegal
- Wrong settings of floating-point flags, memory accesses, and program counter
- Storing the wrong address in the integer register file in case of a misaligned memory access exception.

An example of a non-raised illegal instruction exception bug, was caught while verifying FSQRT instruction behavior. The RISC-V standard requires the "rs2" field of FSQRT to be '0' for the instruction to be valid. In the counterexample we had, an instruction encoded with "rs2"≠0 was wrongly decoded as FSQRT instead of being undefined. Another unexpected bug that this flow discovered centers around memory instructions causing misaligned access exceptions, if they are preceded by a floating-point instruction, and their source register is the same as the destination register of the floating-point one, they store the wrong misaligned address in mtval CSR, which is due to a forwarding issue.

XI. Summary

Exhaustive formal verification is a great technology to add to a complete verification flow to identify late bugs in the process as it strengthens the entire chain of verification analysis by not letting bugs find the direct path to failure. Automating this technology for RISC-V processor verification brings a high degree of efficiency, where direct, actionable feedback leads to shorter debug time for bug fixes and is a great complement to other verification methodologies.

References

[1] https://en.wikipedia.org/wiki/Swiss_cheese_model
[2] https://codasip.com/2022/04/29/building-a-swiss-cheese-model-approach-for-processor-verification/
[3] https://riscv.org/technical/specifications/
[4] https://github.com/riscv/sail-riscv
[5] https://riscv.org/