2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Formal verification of low-power RISC-V processors

Ashish Darbari
Axiomise
71-75 Shelton Street, Covent Garden, London, WC2H 9JQ, England
ashish.darbari@axiomise.com

ABSTRACT

Verification of modern-day processors is a non-trivial exercise, and RISC-V is no exception. In this paper, we present a formal verification methodology for verifying a family of RISC-V "low-power" processors. Our methodology is both new and unique in the way we address the challenges of verification going beyond just functional verification. We focus on architectural verification, lockstep verification (part of functional safety), X-issues due to low-power design, and security. Our approach finds bugs in previously verified and taped-out cores as well as establish bug absence through exhaustive proofs of correctness for end-to-end checks. At Axiomise, we have designed a new RISC-V ISA formal proof kit™ covering RV32IC subset of the RISC-V ISA (so far) to address the problem of architectural verification. We find architectural bugs – ISA instructions not executed in the microarchitecture but also prove bug absence exhaustively when an ISA has been implemented correctly. So far, we found several bugs covering architectural violation, X-propagation, and potential security vulnerability on zeroriscy besides seeing multiple failures with deadlock checks. On ibex, a new 32-bit core under development, so far, we have discovered several failures on architectural checks and eight violations of lockstep verification, and X-issues. Our work defines a new milestone in exhaustive formal verification of microprocessors as it proposes a new way of addressing several verification challenges by combining our new formal tool vendor-neutral ISA formal proof kit along with lockstep verification, deadlock checking, X-checking, and security analysis. Everything we did in our work can be used by any designer or verification engineer using any formal verification tool of their choice. Within minutes of setup, they can be up and running finding bugs as well as having proofs of bug absence and ISA compliance.

I. INTRODUCTION

Functional verification of processors has been a known challenge, but with the advent of open-source RISC-V® and MIPS®, a new wave of computing has ushered. Not only universities, but several big corporations have accelerated the design of new RISC-V cores. The range of applications is varied from the smallest 2-stage pipelined low-power CPUs to bigger out-of-order superscalar designs. One thing, however, is common – verification remains a challenge. Despite all the excitement on new micro-architectural variations from chip vendors, one still needs to demonstrate that the design works as intended. Because the ISA itself is free and open-sourced and in many cases, the design implementations are also open-source, it has become easier for formal verification evangelists to investigate whether these designs do as they say on the tin.

The RISC-V design eco-system is also spanning a new verification eco-system, especially for formal verification with several players now exercising formal verification ranging from theorem proving specific [8] to property checking based solutions [9,10]. Whereas, theorem proving based verification doesn't suffer from the state-space explosion challenge that formal property checking based verification has, the learning curve with theorem proving is significant and is not something widely adopted in the industry at large. Another major weakness of theorem proving is that while one can construct proofs of correctness by the safe and systematic application of inference rules in a theorem prover, one cannot ever obtain a counter-example when there is a bug in the implementation or specification. You almost need a mathematical wizard to discover the problems as part of the proof building process manually.

Formal property checking aka model checking is an automated formal verification technology suited for finding bugs as well as building proofs with several established commercial vendors offering formal tools with a very wide base of users in the industry. The main challenge is in understanding how to develop formal properties that can generate predictable & scalable results across all the formal tools. The industry needs a tool-agnostic solution that can be used by anyone using any formal tool of their choice with guaranteed predictability and scalability.

In this paper, we present a formal verification methodology that provides a tool vendor-agnostic solution for the formal verification of RISC-V processors requiring the user to have no formal verification expertise. With a simple set up, the user can orchestrate a range of verification checks on their RISC-V designs without having to write a single line of the verification code. As our solution is designed using the non-proprietary SystemVerilog Assertions (SVA) supported by all the tool vendors, it makes it easier for designers and verification engineers in the industry to understand our solution, review it, and expand on it if they choose to. Using our kit, users get the results verifying beyond doubt that their design will work as intended against the architectural requirements of the RISC-V ISA. Our methodology also provides checks for finding deadlocks in the design as well as probing the design for any X-related issues typical of low-power designs, while at the same time finding dynamic power issues and discovering any potential security holes as well as establishing baseline compliance for lockstep verification – a requirement for automotive processors.

II. VERIFICATION CHALLENGES

ARCHITECTURAL VS. MICRO-ARCHITECTURAL

A processor implements what its ISA mandates. In addition to compliance with ISA, it is also important to check that the processor works correctly with respect to the microarchitectural specification. What makes the microarchitectural and the architectural verification challenging is that at any point in time, multiple instructions can be in flight competing for the shared resources (e.g., ALU, register files, memory) creating complex dependencies making it easier to introduce design bugs. Specifying a behavior of LOAD instruction at an architectural level is easy – a LOAD instruction loads the data from a data memory from the address computed by adding the offset, and the value from register rs1. Variants of LOAD that load full word, half-word, and byte are also specified in the ISA. However, how a specific implementation will handle misaligned loads (or STORES) is not part of the ISA, and it doesn't need to. This kind of detailed information needs to be specified in a micro-architecture specification, and when it is specified it makes verification easier as there is much less to guess. Without this detailed additional information, it is nearly impossible to do a complete verification of the behavior of LOADS (or STORES). The problem is that in most cases this information is not well-documented and needs to be obtained through discussions with the designer, or through an initial design code review (not ideal).

Timing plays a key role in design but also poses a challenge for architectural as well as micro-architectural verification – determining when to observe the expected values is a key requirement for microarchitectural as well as architectural verification. For example, an ADD instruction in the RISC-V ISA requires the operands in registers **`rs1`** and **`rs2`** to be added and the result stored in the register **`rd.`** The specification itself is abstract and doesn't specify any specific timing. However, how many clock cycles it will take for this result to be reflected in register **`rd`** needs to be determined from a micro-architectural specification as this varies with processor implementation. The result may be updated in the register **`rd`** in one cycle or some cases; it can take several clock cycles.

X-PROPAGATION

The use of low-power design techniques such as the use of un-initialized registers (and explicit X assignments), clock gating, and optimized mux implementations (default branches in one-hot coded mux) introduce microarchitectural bugs that can break the compliance of the design with the ISA. We verified the designs to check if any X-propagation issues existed in them.

DEADLOCKS

To resolve complex dependencies with shared resources and timing, designers use FSMs to track the different design states. However, these FSMs often become the source of deadlock – meaning stuck in a state and cannot come out of it. Besides, the coded FSMs, other control in the hardware is also susceptible to deadlocks. A deadlock checking method should ensure that there is no deadlock in the design at all and should, therefore, verify deadlocks everywhere not just in explicitly coded FSMs. We devised a new way of looking at this problem by focusing on all single-bit signals in the design and asking the question – do these signals toggle infinitely often? It is important to mention that checking that it toggles only once is not enough – they should toggle infinitely often.

SECURITY

Being able to verify, that the design exhibits only those behaviors that the specification mandates and none other is a significant challenge as this requires a very deep and intimate understanding of both architectural and micro-architectural requirements. Grabbing architectural requirements is easy for RISC-V implementations, but finding detailed micro-architectural requirement is very difficult as most of the times they don't exist. From our point of view what we need is a detailed description of all key pipelined units - interfaces, their legal specified behavior (what is allowed and what isn't) and detailed behavior across I/O of each unit.

Being able to decode all this information about the design is necessary to prove that the design implements only specified behaviors. This is a key requirement for checking security. When we started verifying the processors, we didn't have a detailed security specification.

LOCKSTEP VERIFICATION

One of the key requirements for any processor used in automotive and safety-critical environments is to be able to prove that two or more copies of the processor can run in a lock-step manner. It means that if the inputs to multiple copies of the core are identical, the outputs of these processors must not disagree. Verifying exhaustively, that this is the case with any processor is a massive challenge beyond the scope of dynamic simulation but is also challenging for formal verification. However, with good methodology, we can provide exhaustive outcomes with formal verification. This means we can find bugs when the processors produce different outcomes despite their inputs being driven identically, as well as prove that the outputs don't disagree when there are no bugs.

III. PROPOSED SOLUTION

In this section, we present our formal verification approach to address the verification challenges outlined in the previous section. We designed several automated formal verification techniques that were used to address the challenges of (1) architectural verification (2) X-propagation (3) deadlock verification (4) lock-step verification (5) security and (6) dynamic power.

We leveraged standard tool features such as linting, autochecks, tool-built deadlock checking and X-checking but also invested in building our flows independent of a given tool as different formal tools address these problems differently and not all the tools catch all the design issues. Besides, scripting our layer of checking on top of the tool stack means our checks work across all the tools producing the same results, making it easier for DV engineers to compare results. The key aspects of our methodology are outlined here.

1. **Architectural verification**

   We took a transactional approach to model architectural checks in our proof kit. We considered each instruction as a dynamic transaction in hardware with a defined starting point (when it is issued) and an endpoint (when it completes). Mostly, we had one assertion per instruction check (with some exceptions such as JAL and JALR, and LOAD/STORES). The basic methodology relied on capturing the start, and the endpoint of each instruction by sampling expected values and actual value in testbench registers and controlling the sampling through the start point and endpoint. This way we allowed complete freedom to the CPU to issue whatever instruction it wanted to and when it wanted to, but our modeling code will sample in the start point, endpoint, expected value, and actual value whenever an instruction was detected to have been issued and completed. This approach allowed a whole stream of instructions to be in flight at any time during our checking, allowing all kinds of permutations of instructions to be in flight before and after the one we are checking at any point of time. The basic code for each check states that *"when this instruction is issued, then sometime later we expect to see the results of its execution"* having a cause-effect relationship captured in the assertion.

   Consider the following example of checking a BEQ instruction in the RISC-V ISA. The BEQ instruction specification states that if the values in register **`rs1`** and **`rs2`** are equal when a branch-if-equal (BEQ) is detected then the next state of the program counter (PC) will be the branch address. It is modeled in our proof kit through the following SVA.

```
assert property ($rose(is_a_beq) && (axiomise_REG_rs1==axiomise_REG_rs2)
                  |-> ## `REG_DELAY axiomise_n_pc==branch_addr);
```

   Note the use of **`$rose.`** It allows for efficient simulation runs for the same assertion and the use of the macro **`` `REG_DELAY, ``** which specifies for a given implementation how long will it before the PC will be updated. Also, note the use of **`axiomise__*`** prefix, which is used to distinguish the namespace of testbench signals from design signals. The following expressions in Verilog define the signals used in the property.

```
assign branch_addr  = axiomise_decoded_pc +
                                      {{19{axiomise_instr[31]}},
                                        axiomise_instr[31],
                                        axiomise_instr[7],
                                        axiomise_instr[30:25],
                                        axiomise_instr[11:8], 1'b0};
assign is_a_branch  = axiomise_instr[6:0] == AXIOMISE_OPCODE_BRANCH;
assign is_a_beq     = is_a_branch && axiomise_instr[14:12]==3'b000;
```

   We ensured that we didn't make comparisons directly with the branch address signal in the design, but instead had an expected model of it in the test bench derived from the ISA. For verifying the load-store unit, we needed to check all variants of LOADS and STORES both for misaligned cases as well as aligned cases and for byte, half-word, and full-word cases. Our basic approach in modeling these checks was transactional like we explained above but rather than checking all variants of LOAD/STORES in one single property, we broke the problem down to a separate check for each case. The important thing to note, however, is that we allowed complete non-determinism during checking to allow for cases where multiple LOADS and STORES can be in flight for the same as well as different addresses.

2. **X-propagation**

   We wrote checks to verify that none of the design signals was unknown. Rather than writing these checks for all design signals, we wrote them only for signals on the interfaces of the design units. Our view was that if an X cannot reach the interface of a module, it cannot reach the output either, though we did have checks on the output pins as well. One important point on methodology – in formal tools, the checking on a 3-valued

domain is not turned on by default, so one has to explicitly turn-on the X-checking. The treatment of Xs on primary inputs is also different between different tools. For example, some tools force an X on primary inputs as soon as X-checking is enabled, and some don't. So extra care is needed to enforce constraints on primary inputs to force non-X values on them, or else we get spurious failure. If we drive X on inputs, we see X propagating on the outputs.

3. **Deadlock checking**
   For checking deadlocks, we created a library of assertion checks for each single-bit signal in the design (including wires, registers and control states in FSM) and asked the question if they would toggle infinitely often. It was expressed by writing liveness properties.
4. **Security checking**
   For security, we didn't have detailed specifications, so in the absence of such detailed requirements, we explored potential security vulnerabilities by asking questions on design-build quality and architectural compliance. If the design-build quality is robust, it means it is immune to certain malicious code insertions (trojans), and additionally, if the design can provably demonstrate the compliance against the ISA, then it is secure with respect to the ISA. These two analyses do not mean that the design is 100% secure – it only means it is secure against a subset of vulnerabilities. By robust design quality, we mean that there should be no dangling wires, no unused registers, no un-intended bypass logic in the design that can be exploited, and no unintended use of X. Our architectural checks are strong enough to explore any obvious bypasses in the design that can cause unexpected outcomes with end-to-end architectural checks.
5. **Lock-step verification**
   For formally verifying lockstep behavior we created two copies of the cores and tied their inputs together. We then instrumented checks to verify if the outputs of the cores remain equal at all times. This approach is sufficient to catch all lockstep bugs. If the properties on the outputs do not converge easily, we provide proof acceleration by performing a divide-and-conquer approach examining the comparison points at all design interfaces within the core. It does mean that sometimes, we find spurious failures, but it also means we can reach a predictable (yes/no) outcome.
6. **Dynamic Power**
   We investigated a class of failures related to the stability of signals on the interface of the core. We wanted to check if the payload remained stable when it should and doesn't cause any un-necessary toggle of flops resulting in dynamic power consumption.

IV. RESULTS

In the first instance, we started verifying the well-known 32-bit processor zeroriscy from the PULP platform group, which had been previously verified and taped-out at 40 nm in ASIC and FPGA. Within the first six weeks, we found several deadlock bugs as well as exhaustively proved end-to-end functional properties on ISA. A lot of the time spent in this initial phase was spent in building our ISA formal proof kit and understanding/debugging failures. The actual run-time for the checks was a matter of hours but debugging each failure and classifying it as a genuine design bug took time.

zeroriscy

Deadlock: Most of the bugs were down to a buggy debug unit. The debug unit caused 77 deadlock properties to fail out of the 134 checks suggesting a problem. We reported these to the designers, but as the designers already know that there are problems with the debug, they haven't investigated any of these failures. We have done an initial investigation on a few of these and found that if the incoming debug request arrived more than once or if there was a debug resume event after the first debug request was initiated, it causes several design signals to lockup.

Architectural & Security: We found an interesting bug through an architectural check on the LOAD instruction. It is described in more detail in the next section. This bug not only causes problems with functional correctness of the LOAD execution and exposes a potential security vulnerability.

Dynamic Power: Another class of bugs we found were on the instruction and data memory side where stability checks had failed. The core sends out a valid request (`*req_o`) and when an incoming grant (`*gnt_i`) arrives the address and other related payloads such as write data, and byte-enable is transferred across the memory interface. What we noted was that if there is no incoming grant, but there is a valid outgoing request (`req_o==1'b1`), the address, write data and byte-enable was not stable. Our initial response was that these are functional bugs. Designers, however, believed that this is not a real design issue and to save area for a low-power processor, not using registers is a small sacrifice to make. We found that the stability problem does not affect the actual execution of any instruction, so yes, this may not be a functional issue. However, we believe that as addresses, write data, and byte-enable toggle, they burn dynamic power. If for example, the incoming grant is delayed for several cycles on the interface, these signals will continue to toggle burning power – not ideal for a low-power processor. When we conveyed this to the designers, they seem more persuaded to fix this issue.

X-propagation: Other bugs that we found were about redundant undriven logic (causing X issues), redundant, unused signals with constant values, and a potential security vulnerability found due to a failing check on the execution of the LOAD instruction.

Lockstep: We didn't find any bugs with lock-step verification (provided we turn-off the incoming **`debug_req_i`** pin) and were able to exhaustively prove that two copies of the zeroriscy core would run in lockstep.

ibex

Deadlock: For deadlock checking, we found 28 properties that failed the deadlock check, while 40 were proven exhaustively. We didn't investigate all the 28 failures but certainly investigated a few. The tools show us a scenario where it possible to drive an X on the instruction bus infinitely often causing some of the signals to be stuck at 1 or 0. For example, a 32-bit X value can be assigned a combination of 0s and 1s so that it causes the opcode to remain a STORE (or LOAD) forever, causing a data write signal to remain high or low forever. Although we could call this scenario somewhat pathological, it is certainly possible.

Architectural: For ibex, the results are different from zeroriscy, and that is to be expected as the core is still under development, although ibex has been cloned from zeroriscy. We wanted to test our ISA formal proof kit initially on ibex, so we started initially with just architectural testing in mind. It took 30 minutes to set up our proof kit. Our initial set up was to turn off the debug completely (disable the incoming **`debug_req_i`** pin) as we thought ibex was a clone of zeroriscy with now a known debug issue it would be wise to constrain the debug interface off. Once the debug was disabled, we started getting exhaustive proofs of correctness for all the ISA checks in our proof kit. 75% of these were proven in about 7 minutes of wall clock time, while the remaining 10% in 30 minutes and the last 15% in about 12-24 hours depending upon the tool being used. The longest-running proofs are checks that establish that STORE instructions correctly update the memory – correct data is written to the correct address. However, once we enable the **`debug_req_i`** pin to go high by disabling our constraints on the debug interface, we noticed that almost all the ISA checks (about 57 of our ISA properties) failed. We were puzzled to see these many failures. It turned out that almost all of them are down to one major bug in the design related to a controller state machine. When the state machine was in the DECODE state and an input debug request was made in that cycle; it will end up failing the execution of the instruction that was in flight (being decoded and executed). We will provide more analysis on this in the next section. We didn't see the REG-REG load problem on ibex, as the buggy code has been removed in ibex after we reported this on zeroriscy.

X-propagation: We tested ibex for X-propagation and found eight bugs, regardless of whether the debug interface is enabled or not. It means that the processor will exhibit correct functionality for all instructions we tested when debug is disabled, but it will show reachable X issue for some of the outputs of the core. When a design runs in silicon, there is no X, only 0s and 1s. The X that cause differences in lockstep behavior does not cause any problems during the ISA execution. It is because an X is treated like a Boolean during a normal formal verification run, allowing for the X to be replaced by 0 and 1.

Lockstep: We found eight violations for lockstep. The root cause of this is X-assignments. If two cores are configured to run in lockstep, they exhibit different I/O behavior, as the X in the design takes on the Boolean value 0 in one copy of the design and 1 in the other copy.

Security: It is well-known that if an X is reachable at the outputs, it can cause information leakage if an attacker can observe the primary inputs [17]. What we see with the X-failures and lockstep analysis – visible Xs and lockstep differences on outputs exposes potential security vulnerabilities with ibex. Our deadlock analysis revealed that due to the X-problem, some of the outputs and internal state could be deadlocked; this happened as X on the instruction bus. Though it seems pathological at first, we believe an attacker can make use of these input scenarios to cause the core to lockup.

Dynamic Power: The dynamic power issue we noted on zeroriscy is still there on ibex.

V. BUG ANALYSIS

Although we found several different kinds of bugs, we do not have adequate space in this paper to describe them all. In this section, we describe two example architectural bugs – one with zeroriscy and the other from ibex.

1. Architectural violation: ibex (BEQ instruction)

We used the architectural check shown earlier to find this bug. The bug is caused due to the incoming debug request when the controller state machine is in DECODE state. Because there is nothing in the ibex design to prevent this scenario, it causes the execution of BEQ to be interrupted and causes the PC to be not updated correctly. On a closer investigation, we found that this bug will only occur if the precise arrival of incoming debug request happens to be when the controller FSM is in DECODE, there is no problem if the debug arrived any time before or later. We confirmed this by controlling the arrival of debug – prevent it from arriving when the controller FSM is in the DECODE state. On assuming this property, the failing property became an exhaustive proof. We are not sure if this is entirely deliberate from a design point of view. We believe in general it may not be possible to control the arrival of incoming debug

request to only happen when the controller FSM is not in the DECODE state, so this looks like a real design issue. On investigating the design code, it looks like the designers intend to implement some control logic to avoid these buggy scenarios as evidenced from some comments left in the design (line 268 in the ibex_controller.sv file). As this problem affects all the other instructions in ibex as well, we got intrigued to see why this is not occurring in zeroriscy the predecessor of ibex. We found that in zeroriscy, the designers have put appropriate logic to take care of this scenario, which is missing from ibex. What is very interesting about this bug is that though it was found very easily by our ISA formal checks, it will be very difficult if not completely impossible to find this in simulation.

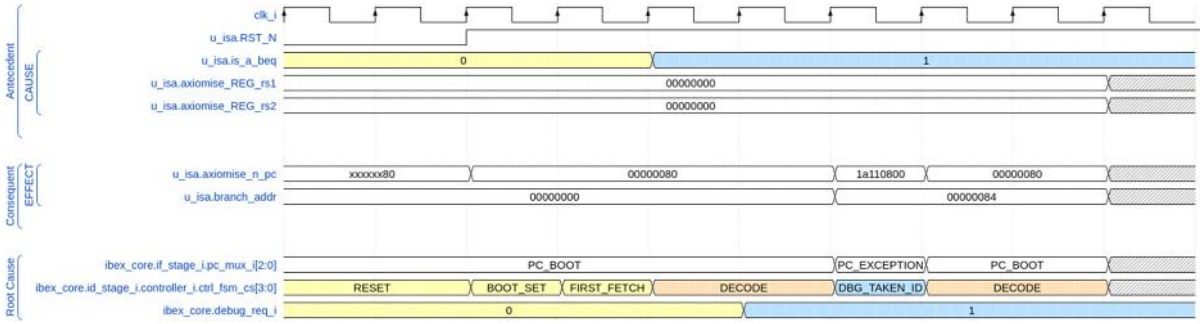

Fig 1. BEQ failure on ibex, showing the bug we found using Axiomise ISA formal proof kit™.

The reason is that to come up with a precise stimulus in simulation to drive **`debug_req_i`** high when one of the states in one of the many FSMs in the design happens to be in **`DECODE`** state is going to be very difficult. In the case of ibex, like other processors, we have several interacting FSMs and creating random seeds in simulation to cover all states in all FSMs for all inputs is not feasible.

2. Architectural violation: zeroriscy REG-REG LOAD

The following code shows the bug where the unintended REG-REG load code is affecting the expected LOAD behavior.

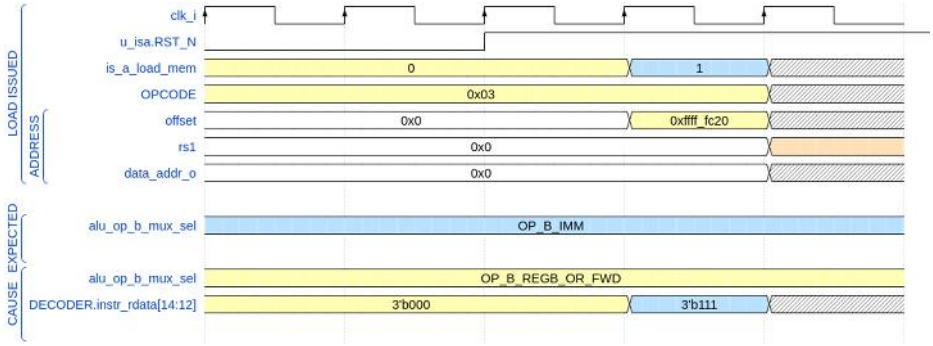

Fig 2. zeroriscy: Bug in the LOAD instruction implementation.

The load address computation check fails shown below where one of the operands needed for the address sum is coming from the **`OP_B_REG_B_OR_FWD`** signal as opposed to the **`OP_B_IMM`** field, which is what the ISA mandates. As part of checking the behavior of LOAD instruction, we wrote a check to see if the address computed by the LOAD instruction is correct.

```
logic [31:0] load_offset;
assert property ($rose(is_a_load_mem) |->  `LSU.data_addr_` == rs1 + load_offset);
assign is_a_load_mem  = `DECODER.instr_rdata_i[6:0] == OPCODE_LOAD;
assign load_offset = !data_misaligned_o ?
                          {{20{`DECODE.instr[31]}},`DECODE.instr[31:20]} : 32'h4;
assign rs1         = !data_misaligned_o ?
                          `REGISTER.mem[`DECODER.instr_rdata_i[19:15]] :
                          `DECODE.misaligned_addr_i;
```

LOAD is an I-type instruction in the RISC-V ISA and it mandates that the immediate offset is computed from the bits **[31:20]** of the decoded instruction with 20 bits of MSB padded with bit 31 of the decoded instruction, but in this case, the designers have guarded the offset to be the architectural case only for aligned addresses. For misaligned addresses the offset is different. Expected code that correctly sets the MUX input (**alu_op_b_mux_sel_o**) to choose the immediate input **OP_B_IMM.**

```
OPCODE_LOAD: begin
...
alu_op_b_mux_sel_o     = OP_B_IMM;
imm_b_mux_sel_o        = IMMB_I;
```

However, a few lines later in the design file we find the buggy code in the decoder module which causes the **alu_op_b_mux_sel_o** to take on a different value because the designers have coded in a REG-REG load not part of the RISC-V ISA.

```
if (instr_rdata_i[14:12] == 3'b111) begin

    // offset from RS2
    alu_op_b_mux_sel_o = OP_B_REGB_OR_FWD;

    // sign/zero extension
    data_sign_extension_o = ~instr_rdata_i[30];

    // load size
    unique case (instr_rdata_i[31:25])
      7'b0000_000,
      7'b0100_000: data_type_o = 2'b10; // LB, LBU
      7'b0001_000,
      7'b0101_000: data_type_o = 2'b01; // LH, LHU
      7'b0010_000: data_type_o = 2'b00; // LW
      default: begin
        illegal_insn_o = 1'b1;
      end
    endcase
  end
end
```

The problem with this bug is not just that the address computed is wrong which is a significant issue anyway, but that LOAD instructions decoded in compliance with the RISC-V ISA will not transfer the correct data payload from the memory to the register if another LOAD instruction (REG-REG LOAD) was also in flight with the same address. It affects all the variants of LOAD checks for byte, half-word, and full-word and cases for aligned and misaligned. We believe that though this may be unintended and not necessarily a malicious code insertion, it does exhibit a security vulnerability where an undesirable bypass alters expected code execution on memory reads by the CPU.

VI. RELATED WORK

Although formal verification of microprocessors is not a new domain [1-7], and neither is formal verification of RISC processors [2], the application of formal verification to RISC-V is relatively new [8,9,10]. The earliest applications made use of theorem provers, and the emphasis has largely been on establishing functional correctness of processor models designed in the language of the underlying proof tool. Ken McMillan devised advanced problem reduction techniques and showed how out-of-order processor implementations could be verified with formal when the processor design was coded in the formal tool SMV [4]. Widespread applications of formal verification at the commercial level have largely been limited with notable exceptions being the pioneering work done by Boyer Moore who verified the floating-point unit of AMD® processor using ACL2 theorem prover [3], Intel® using their proprietary tool Forte [5], IBM® using its proprietary tool Rulebase Sixth Sense [6], and more recently Arm® [7] who reported developing a tool to verify ISA compliance of their implementations. Alglave et al.'s work [16] on relaxed memory models and our work [15] on using Event-B for formal modeling, testing and verification of HSA memory models are other important pieces of work in the context of formal methods and processor verification.

In the context of RISC processors, S. Tahar's work on formal verification of the first generation of RISC processors [2] is important. He devised a framework of correctness and formalized it inside the HOL 4 theorem prover. For RISC-V, a lot of effort has kicked-off in applying formal verification. On the one hand, formal tools such as Coq and Bluespec are used to model the ISA of RISC-V [8] with the intent of finding consistency issues; and on the other hand property checking based solutions have been used for finding micro-architectural bugs as well as establish conformance to the ISA [9, 10]. Whereas the work done by Chilipala in [8] is indeed valuable, it is not clear how many users are familiar

with Coq and BlueSpec. A lot of hardware design is done using standard VHDL/Verilog, and it is not clear how the work that is done in [8] can be of direct use in verifying such designs without a substantial investment in enabling people in writing processor designs in BlueSpec, and at least having a basic understanding of Coq formalisms. We believe that the work by Hummenberger et al. in [9] and the OneSpin Solutions® [10] appears to be directly valuable and useful for a vast majority of designers and verification engineers building RISC-V designs. However, both solutions require the end-user to be locked into the respective proprietary tools. Although Hummenberger et al. [9], do have an open-source version of their tool in the public domain, this requires an intrusive setup requiring the instrumentation of the RISC-V design with testbench interface signals and requires modifications to suit the tool limitations as evidenced by some of their users [11]. The solution from OneSpin is developed around a OneSpin proprietary assertion language (called operational assertions), and there are claims that by using this they can guarantee a gap-free formal verification solution. However, to exploit the gap-free solution, the user needs to use the formal tools (DV-Verify® and DV-Certify®) from OneSpin – as there is no support for operational assertions inside any other formal tool. OneSpin does mention the usage of SystemVerilog assertions (SVA) and that there is a translator for operational assertions into SVA – it has been made clear through their publications [12] that to obtain gap-free solution one must use operational assertions not standard SVA to model requirements.

VII. CONCLUSION

Our reusable methodology provides unique insights on verifying a family of RISC-V processors, some of whom have been previously verified and taped-out. We found bugs (including corner case ones) and proved conclusive bug absence when we fixed bugs. Although we used QuestaFormal® for most of our work, our solution is completely vendor agnostic, can be run with any formal tool. Run time performance varies between tools, but the result is identical for determined Pass/Fail outcomes. Our solution doesn't require any special skills or to know a specific tool [10,12] or familiarity with a domain-specific language [8,10,12]. We have used the well-known general-purpose languages – Verilog and SVA supported by all the formal tool vendors. We believe this is a significant milestone – being able to prove end-to-end functional correctness and finding bugs in 32-bit processors using vendor-neutral formal tools, using our automated architectural (ISA) formal proof kit augmented with other important verification tasks such as deadlock and lockstep verification and X-propagation.

REFERENCES

[1] A Proof of Correctness of the VIPER Microprocessor, *VLSI Specification, Verification, and Synthesis*, A. Cohn, 1988.

[2. A Practical Methodology for the Formal Verification of RISC Processors, S. Tahar et al., *Formal Methods in System Design*, 1998.

[3] A Mechanically Checked Proof of the AMD5K86TM Floating-Point Division Program, J. Moore et al., *IEEE Transaction on Computers*, 1998.

[4] Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking, K. McMillan, *Proceedings of CAV*, 1998.

[5] Practical Formal Verification in Microprocessor Design, R. Jones et al., *IEEE Design and Test of Computers*, 2001.

[6] Integrating FV into Main-Stream Verification - The IBM Experience, J. Baumgartner, 2006.

[7] End-to-End Verification of ARM Processors with ISA-Formal, A. Reid et al., *Proceedings of the CAV*, 2016.

[8] Strong Formal Verification for RISC-V, A Chilipala, RISC-V Workshop, 2017.

[9] Formal Verification of RISC-V Processor Implementations, Humenberger et al., RISC-V Summit, 2018.

[10] Complete Formal Verification of RISC-V processor IP for Trojan-Free Trusted ICs, *GOMAC*, 2019.

[11] Synthesising Ibex with Yosys, Github Note., https://github.com/lowRISC/ibex/issues/60

[12] Complete Formal Verification of TriCore2 and other processors, J. Bormann et al., *DVCon* 2007.

[13] Slow and steady wins the race? A Comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. P D. Schivaone et al. *International Symposium on Power and Timing Modelling, Optimization and Simulation (PATMOS), 2017*.

[14] lowRISC/ibex. https://github.com/lowRISC/ibex

[15] Formal Modelling, Testing and Verification of HSA memory models using Event B, A. Darbari et al., *ArXiV*, 2016.

[16] The Semantics of Power and ARM Multiprocessor Machine Code, Alglave et al., *DAMP* 2009.

[17] Hardware IP Security and Trust, P. Mishra et al., Springer Verlag, 2017.