Formal Verification of the Stall Invariant Property for Latency-Insensitive RTL Modules

Peitian Pan, Christopher Batten
Cornell University
Ithaca, NY, USA
{pp482,cbatten}@cornell.edu

ABSTRACT

Latency-insensitive protocols are widely used in hardware standard libraries and network-on-chip IPs because they enable modular hardware design and efficient circuit implementation of communication channels. However, RTL modules with latency-insensitive protocols at their interfaces (or *latency-insensitive RTL modules*) create a verification challenge because subtle design bugs in these RTL modules may only be triggered after a specific number of stall cycles on the latency-insensitive interfaces. Verifying latency-insensitive RTL modules with simulation-based techniques requires a comprehensive test suite that covers all possible stall cycles up to a sufficiently large number, which needs significant verification efforts to build and maintain. In this paper, we propose a formal verification methodology to detect bugs in latency-insensitive RTL modules by verifying the stall invariant property of these modules. We introduce bounded latency equivalence checking (BLEC) to detect violations of the stall invariant property under finite buffering. BLEC includes a systematic approach to construct a verification harness which applies ingress and egress stalls and checks if the DUV egress results are the same under varying stall conditions. We implement the proposed method with state-of-the-art commercial formal verification tools and demonstrate its effectiveness with case studies on a latency-insensitive processing element, a greatest common divisor unit, and a pipelined RISC-V processor. In all three case studies, our proposed method can detect subtle design bugs inserted in the design. With some manual simplifications to the target RTL modules, existing formal verification tools can provide a bounded proof of the stall invariant property to many RTL modules.

CCS CONCEPTS

• **Hardware → Equivalence checking**.

KEYWORDS

formal verification, latency-insensitive designs

ACM Reference Format:
Peitian Pan, Christopher Batten. 2023. Formal Verification of the Stall Invariant Property for Latency-Insensitive RTL Modules . In *21st ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '23), September 21–22, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3610579.3611081

MEMOCODE '23, September 21–22, 2023, Hamburg, Germany

ACM ISBN 979-8-4007-0318-8/23/09…$15.00
https://doi.org/10.1145/3610579.3611081

1 INTRODUCTION

Latency-insensitive (LI) protocols [8, 10, 11] are an effective hardware design methodology that significantly improves design productivity with minimal performance, power, and area overhead [9, 19]. By decoupling the communication and computation aspects of hardware design, RTL modules with latency-insensitive interfaces (or simply latency-insensitive modules) offer two major benefits over the traditional synchronous design paradigm [5, 17]. First, hardware designers can safely compose modules with latency-insensitive interfaces without worrying about the potentially *variable* latencies of upstream and downstream modules. In the case where the upstream or downstream modules are not generating valid messages or not ready to accept messages, a *stalling* event occurs on the latency-insensitive interface and sequential states are preserved until an *informative* event containing the real message eventually happens [11]. Second, latency-insensitive protocols enable more efficient circuit implementation of communication channels than with the synchronous design paradigm. Inter-module communication channels designed under a synchronous system assumption often synthesize into long global wires that limit the system clock frequency. On the other hand, the communication between latency-insensitive modules can be pipelined by inserting *relay stations* [11] between the modules to achieve higher clock frequency. Because of these benefits, latency-insensitive modules are virtually ubiquitous across hardware standard libraries, hardware compositions, and network-on-chip IPs [4, 15, 16, 20, 22, 23].

However, implementing latency-insensitive RTL modules presents a unique verification challenge. Figure 1 shows a two-stage pipelined latency-insensitive RTL module. To handle the potential backpressure from the egress interface or the input delays on the ingress interface, the design under verification (DUV) has complex control logic which includes pipeline register enable signals, per-stage originating stall signals, and a squash signal. Two examples of control logic bugs are also in this figure. For Bug-A, the `ostallA` signal is not propagated to the enable signal of stage B (`enB`), which means pipeline register `B` can be enabled while stage A is originating a stall and may register incorrect data. For Bug-B, the `ostallB` signal is not propagated to the control logic of stage A, which can lead to data loss because the content of pipeline register `B` can be overwritten by outputs from stage A even when stage B is originating a stall. It is worth noting that these two bugs only manifest when there is backpressure on the egress interface, and that similar and subtler bugs might only get triggered with a specific number of cycles of stalls on the ingress and/or the egress interface. It is

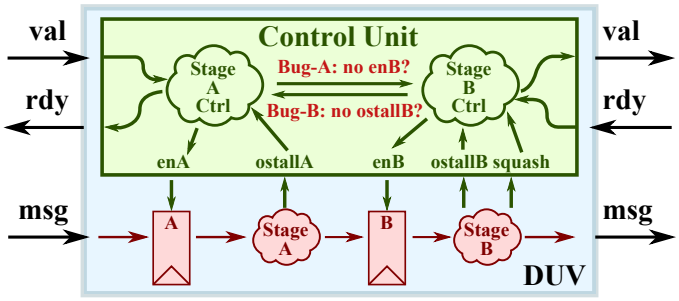


Figure 1: Examples of Bugs in a Two-Stage Pipelined Latency-Insensitive RTL Module – the design under verification (DUV) has complex control logic to adapt to possible delays on the ingress (left) or the egress (right) interface. Both pipeline registers (`A` and `B`) are enabled when there is no stall originating from their respective stages; stage B can also *squash* stage A due to a hazard that is only visiable in a later stage. Bug-A is a design bug where the `ostallA` signal is not accounted for in the enable signal of stage B (`enB`); Bug-B shows a bug where the `ostallB` signal is not propagated from stage B to stage A. `ostallA/B`: signal is asserted if stage A/B originates a stall; the `ostall` signals may be propagated to earlier stages; `enA/B`: enable signal for pipeline register of stage A/B.

challenging to discover these bugs via simulation-based dynamic verification techniques. Detecting these bugs in simulation needs a comprehensive test suite that covers all possible stall cycles (up to a sufficiently large number) on the DUV's latency-insensitive interfaces, which requires significant testing and verification efforts to build and maintain.

In this paper, we propose a formal verification methodology to address the verification challenges of latency-insensitive RTL modules. We make the observation that most correct latency-insensitive RTL modules have the same behavior even under different number of stall cycles, which we call the *stall invariant* property. We propose bounded latency equivalence checking (BLEC), a technique that detects violations of the design under verification (DUV)'s stall invariant property under finite buffering. BLEC constructs a verification harness that contains two duplicated DUVs with different stall conditions and verifies the latency equivalence [10] between the DUVs using formal verification. A BLEC verification process generates one of two possible outcomes: (1) BLEC finds a violation to the stall invariant property of the DUV and provides a waveform to help identify origin of issues or (2) BLEC proves that the stall invariant property holds true for the DUV up to a certain number of stall cycles.

This paper makes the following contributions:

- we introduce the stall invariant property and make the observation that many bugs in latency-insensitive RTL modules violate the stall invariant property;
- we propose bounded latency equivalence checking, a formal verification technique to detect violations of the DUV's stall invariant property under finite buffering; we implement bounded latency equivalence checking using state-of-the-art commercial formal verification tools;
- we demonstrate the effectiveness of our proposed method by evaluating bounded latency equivalence checking on three latency-insensitive RTL designs: a latency-insensitive processing element, a greatest common divisor (GCD) unit, and a RISC-V pipelined processor.

2 THE STALL INVARIANT PROPERTY OF LATENCY-INSENSITIVE RTL MODULES

In this section, we introduce the stall invariant property with the motivating DUV in Figure 1. Figure 2 (a)-(c) refer to the behaviors of the DUV without bugs, with Bug-A, with Bug-B, respectively. We examine the behaviors of the DUV both with and without bugs and compare the behaviors under different stall conditions. We make the observation that bugs in RTL modules generally lead to inconsistent behaviors on the egress LI interface under different ingress and/or egress stalls.

Events in Latency-Insensitive RTL Modules – The behaviors of each design in Figure 2 are characterized by the sequence of *events* that occurs on the ingress and egress LI interfaces of the DUV. Using the terminology from the original latency-insensitive design theory paper [11], we call events where a message is successfully transferred over the LI interface an *informative event* (cycles that are marked a, b, c, or × in Figure 2); we call any other events where a message is not transferred *stalling events*. We further classify stalling events into two categories: (1) not-valid (indicated by symbol – in Figure 2), where the sender of the LI interface is not valid to send a message at the cycle of the event; (2) not-ready (indicated by symbol # in Figure 2), where the sender of the LI interface has valid message to send but the receiver is not ready to accept that message at the cycle of the event. In an RTL module that implements a val-rdy LI interface (e.g., DUV in Figure 1), not-valid stalling events correspond to cycles where `val` is low and not-ready stalling events are cycles where `val` is high but `rdy` is low.

Stall Conditions of a Latency-Insensitive DUV – The ingress and egress interfaces of a LI DUV need to be connected to upstream and downstream modules for the DUV to function properly. An upstream module can apply *input stalls* to the DUV by de-asserting the `val` signal at cycles it does not have valid messages to send, which creates a not-valid stalling event. Similarly a downstream module can apply *output stalls* to the DUV by de-asserting the `rdy` signal at cycles it is not ready to accept messages from DUV, which can create a not-ready stalling event. For the same sequence of informative events, we call the cycles where input and output stalls are applied the *stall condition* of the DUV. To make our explanations more clear, we examine three simple stall conditions for each design in Figure 2: (1) No Stalls, where the ingress interface is always valid to send a message to the DUV and the egress interface is always ready to accept a message from the DUV; (2) Ingress Stall, where the ingress interface is not valid at cycle 2 for illustration purposes; (3) Egress Stall, where the egress interface is not ready at cycle 3.

Behaviors of the DUV – Figure 2 (a) shows the behaviors of the correct DUV under the three stall conditions described above. It is straightforward that the correct design exhibits pipeline behaviors between its input stage A and output stage B: it always takes two cycles for a message to traverse from ingress to egress when no stalls are applied; applying input stalls creates bubbles in the pipeline, as shown by the not-valid stalling events in cycle 2 and 3 (Ingress Stall); and applying output stalls stalls the pipeline as shown in cycle 3

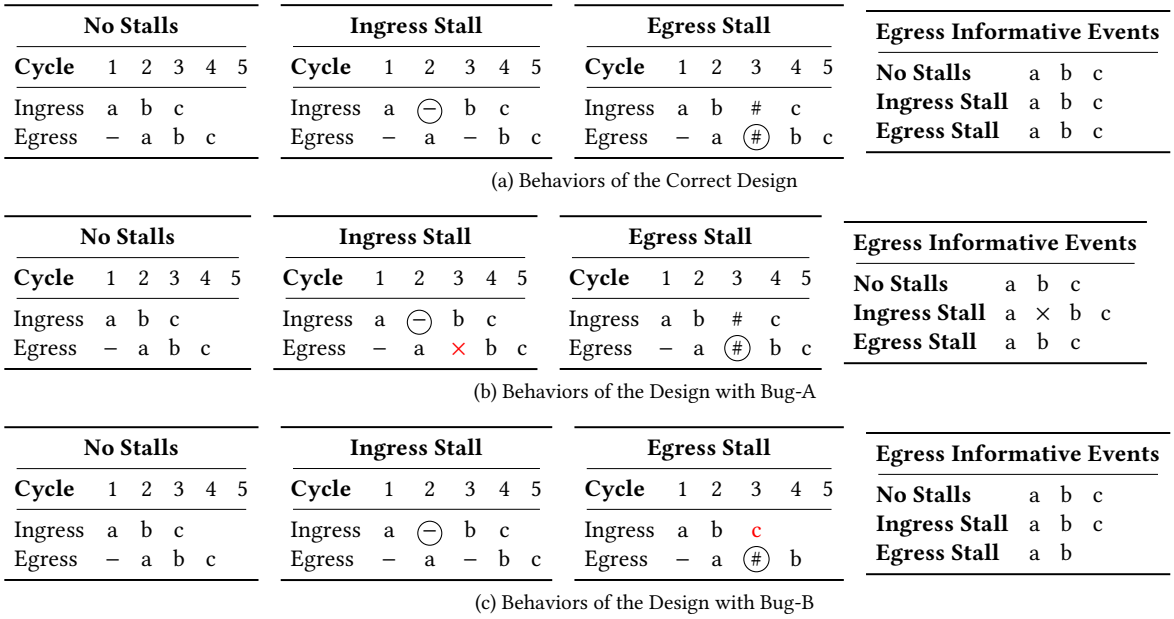


Figure 2: Behaviors of Designs in Figure 1 under Different Stall Conditions – –: not-val stalling event; #: not-ready stalling event. Three designs are used in this figure: the correct design as shown in Figure 1, the design with Bug-A (wrong `enB` signal), and the design with Bug-B (`ostallB` signal not forwarded to stage A). Three stall conditions are used in this figure: no stalls: the ingress LI interface is always valid to produce a message and the egress LI interface is always ready to accept an output message; ingress stall: the ingres LI interface is not valid at cycle 2 (marked with black circle) which leads to a bubble (– at cycle 3) in the correct design's pipeline; egress stall: the egress LI interface is not ready at cycle 3 (marked with black circle) which causes the pipeline in the correct design to stall (# at cycle 3). Only the correct design is stall-invariant because the other two designs have a different sequence of egress informative events (×: incorrect value registered; c: message c accepted when pipeline stalls) either under ingress stalls (Bug-A) or egress stalls (Bug-B).

(Egress Stall). Figure 2 (b) shows the behaviors of the DUV with Bug-A, where the pipeline registers of stage B can still be enabled even when stage A is stalling. Bug-A has the same behaviors as the correct design when no stalls or only output stalls are applied because stage A is not stalled in these two cases; however, when ingress stall is applied on cycle 2, the pipeline registers of stage B will register invalid data from the stalled stage A, which leads to an erroneous output message on cycle 3 (marked by red ×). Figure 2 (c) shows the behaviors of the DUV with Bug-B, where stage A is not stalled when stall B is stalled. Bug-B has the same behaviors as the correct design when no stalls or only input stalls are applied because stage B is not stalled in these two cases; however, when egress stall is applied on cycle 3, message C is lost at this cycle because stage A does not stall.

The Stall Invariant Property – The example in Figure 2 shows that some designs have inconsistent behaviors (as determined by the sequence of informative events on their egress interfaces) under different stall conditions. We call a latency-insensitive RTL module *stall invariant* if the module has the same sequence of informative events on its egress interfaces under all possible stall conditions. The stall invariant property is useful for catching bugs that lead to a different sequence of informative events on the LI interface of the DUV, which include numerous subtle bugs especially in a pipelined DUV module. It is worth noting that the stall invariant property only requires the equivalence of the sequence of egress informative events and does not imply functional correctness of the DUV.

3 BOUNDED LATENCY EQUIVALENCE CHECKING

In this section we introduce bounded latency equivalence checking (BLEC), a formal verification technique that detects violations of the DUV's stall invariant property under finite buffering. For a given latency-insensitive RTL module (the DUV), BLEC constructs a verification harness with formal assertions that can be verified by hardware formal property verification (FPV) tools. The FPV tools can either find a violation of the stall invariant property (which generally indicates the existence of a design bug) or provide a potentially bounded proof that the target DUV is stall invariant. We first introduce the necessary verification modules that are used in the BLEC verification harness (Section 3.1). We then propose a systematic method that constructs the BLEC verification harness for any given latency-insensitive RTL module (Section 3.2).

3.1 Verification Modules

Figure 3 shows the verification harness of a DUV with one ingress and one egress latency-insensitive interface. The verification harness in Figure 3 exposes five input and output ports:

- `val`, `rdy`, and `msg`: these three ports form the LI interface that generates input messages to the ingress LI interface of the DUV.
- `stall_ingress` and `stall_egress`: these two ports are *stall variables* whose value decides if an ingress stall or an egress

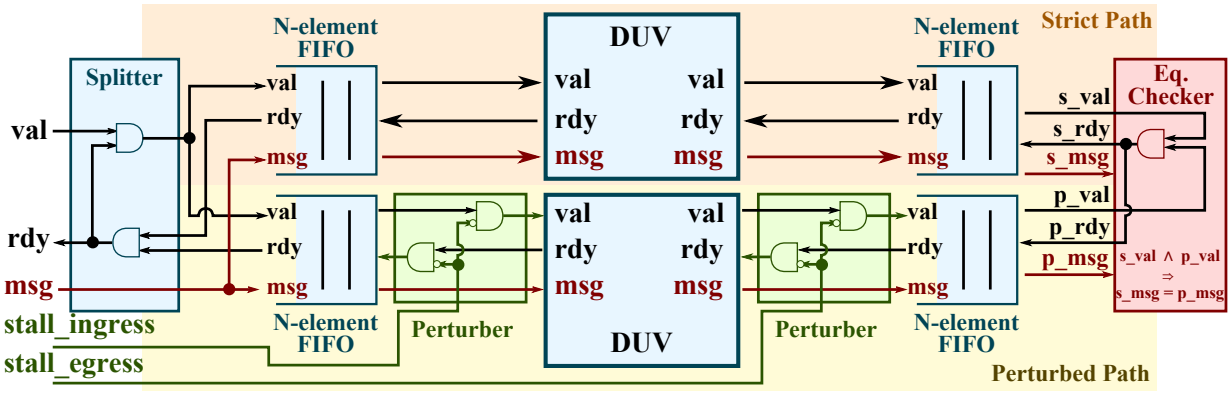


Figure 3: Verification Harness for a DUV with One Ingress and One Egress LI Interface in Bounded Latency Equivalence Checking – `stall_ingress/egress`: stall variables for the ingress/egress LI interface. `N`: a parameter which detemrines the depth of FIFOs in the verification harness. Strict path: a path in the harness where no ingress or egress stalls are applied on the DUV. Perturbed path: a path in the harness where the *perturbers* apply a random number of stalls on the ingress and egress interface. `Eq. checker`: equivalence checker; a module that checks if the result messages from the two paths are the same.

stall is applied on the DUV's LI interface (1 for stall and 0 for not stall).

As is shown in the figure, the verification harness contains two duplicated instances of the target DUV with different stall conditions: the DUV in the *strict path* (i.e., the *strict DUV*) has no ingress or egress stalls under with FIFOs of N elements; the DUV in the *perturbed path* (i.e., the *perturbed DUV*) has random ingress and egress stalls injected by *perturbers*. At the end of both path, a *equivalence checker* compares the result messages in the output FIFOs and reports a violation of the stall invariant property if the two messages are different.

N-Element FIFOs – The verification harness includes four N-element FIFOs to decouple the LI interfaces of the two DUV instances, where N is a constant determined ahead of the construction of verification harness. Two FIFOs are inserted between the ingress LI interfaces of the two DUVs and the top-level ingress LI interface (`val`, `rdy`, `msg`). These FIFOs decouple the strict DUV from the ingress stalls of the perturbed DUV, which achieves almost zero ingress stalls for the strict DUV. Similarly, the two FIFOs between the egress LI interfaces of the DUVs and the message checker decouple the strict DUV from the egress stalls of the perturbed DUV, which achieves almost zero egress stalls for the strict DUV.

Assuming no ingress stalls nor egress stalls are applied on the DUV, FPV tools can generate a proof that the DUV is indeed stall invariant. This can be shown by comparing the behaviors of the perturbed DUV against the strict DUV: the equivalence checker ensures that the sequence of egress informative events of the DUV under all stall conditions (output of the perturbed path) is the same as if no stalls are applied (output of the strict path); therefore, the DUV is stall invariant by definition (Section 2).

It is worth noting that even with deep FIFOs (large N's), the strict DUV may still experience ingress or egress stalls. The FPV tools can still prove that the perturbed and strict DUV have the same sequence of egress informative events. We call this proof a *bounded* stall invariant proof because the strict DUV experiences ingress and/or egress stalls due to finite buffering. The finite buffering also defines the bounded nature of our proposed BLEC technique: BLEC is only able to provide bounded stall invariant proofs because FIFO sizes are finite. The finite size of FIFOs does not affect our technique's effectiveness in finding violations of the stall invariant property because stall variant DUVs mostly generate different sequences of egress informative events under non-zero stalls, not necessarily zero stalls. Large depths of FIFOs may also have negative impacts on the performance of the formal property verification tools. Therefore, we choose a small FIFO depth of two (2) in this paper to decouple the strict and perturbed DUVs without causing too much tool performance overhead.

Perturbers – Perturbers are a verification module inserted between the DUV and FIFOs to inject random stalls to the ingress or egress LI interface of the DUV (i.e., to perturb the DUV with random ingress or egress stalls). A perturber takes as input a stall variable (`stall_ingress` and `stall_egress` input ports in Figure 3), which decides if stall is applied on the LI interface. As the green-shaded components in Figure 3 show, the perturbers connect the `val` and `rdy` LI handshake signals and the corresponding negated stall variable with an AND gate. This logic suppresses the LI handshake (and thus stalls the LI interface) when the stall variable is high.

Equivalence Checker – The equivalence checker is a module that checks if the results of the egress latency-insensitive interface from the two paths are the same. As shown in Figure 3, the checker (in red) is interfaced to the two egress FIFOs. The equivalence checker only dequeues from the FIFOs and performs the equivalence check if both FIFOs are non-empty (i.e., `val` is asserted). The behaviors of the equivalence checker can be expressed as a property of an RTL module, which is boolean expressions between its signals. In Figure 3, we use the implication operator ($\Longrightarrow$) to indicate that the equivalence check between `s_msg` and `p_msg` only happens when both `s_val` and `p_val` are true.

3.2 Construction of Verification Harness

We demonstrate the verification harness of a DUV with one ingress and one egress LI interface in the previous section. In this section,

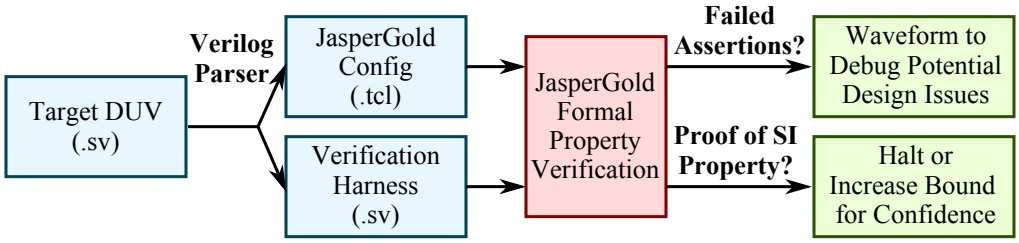


Figure 4: Workflow with Our Implementation of BLEC – SI: stall invariant. The Verilog parser implements Algorithm 1 to generate the verification harness and necessary JasperGold configuration scripts. The generated verification harness contains assertions that JasperGold FPV proves or finds counterexamples to.

Algorithm 1 Construction of the BLEC Verification Harness – s and p in subscripts indicate the module belongs to the strict/perturbed path; i and e in subscripts indicate the signal or module is associated with the ingress interface i or the egress interface e. $TopLI_i$: toplevel latency-insensitive interface that generates messages to the LI interface i. N-FIFO, EqChecker: N-element FIFOs, equivalence checkers as introduced in Section 3.1. H is a set of modules; H_p is a set of interfaces and ports; H_c is a set of tuples where neighboring tuple elements are connected and data flow through elements in ascending index order.

Require: D: The target design under verification.
Require: N: The depth of FIFOs in the verification harness.
Ensure: Verification harness H with ports H_p and connections H_c.

1: **function** ConstructHarness(D, N)
2: $H \leftarrow D_s \cup D_p$
3: $H_p, H_c \leftarrow \emptyset$
4: **for all** $i \in$ IngressLatencyInsensitiveInterface(D) **do**
5: $H \leftarrow H \cup$ N-FIFO$_{s,i} \cup$ N-FIFO$_{p,i} \cup$ Perturber$_i$
6: $H_p \leftarrow H_p \cup TopLI_i \cup$ StallVariable$_i$
7: $H_c \leftarrow H_c \cup (TopLI_i,$ N-FIFO$_{s,i}, D_s)$
8: $\cup\, (TopLI_i,$ N-FIFO$_{p,i},$ Perturber$_i, D_p)$
9: $\cup\, ($StallVariable$_i,$ Perturber$_i,)$
10: **for all** $e \in$ EgressLatencyInsensitiveInterface(D) **do**
11: $H \leftarrow H \cup$ N-FIFO$_{s,e} \cup$ N-FIFO$_{p,e} \cup$ Perturber$_e$
12: $\cup$ EqChecker$_e$
13: $H_p \leftarrow H_p \cup$ StallVariable$_e$
14: $H_c \leftarrow H_c \cup (D_s,$ N-FIFO$_{s,e},$ EqChecker$_e)$
15: $\cup\, (D_p,$ Perturber$_e,$ N-FIFO$_{p,e},$ EqChecker$_e)$
16: $\cup\, ($StallVariable$_e,$ Perturber$_e,)$

we describe a systematic method to construct a verification harness for any latency-insensitive RTL modules.

Algorithm 1 shows the steps to construct a BLEC verification harness for any given latency-insensitive RTL DUV D with N-element FIFOs. The algorithm proceeds by enumerating all ingress and egress LI interfaces of D and adds modules, ports, and connections to the verification harness H. For each ingress LI interface i of D, the algorithm adds one toplevel LI interface to generate messages to i, one perturber to apply random stalls on i, and two N-element FIFOs; for each egress LI interface e of D, the algorithm similarly adds two FIFOs, one perturber, and one equivalence checker to compare the results of the strict and perturbed paths. The generated verification harness H may have multiple equivalence checkers and a violation of the stall invariant property is found if the FPV tool finds a failed assertion in any of these checkers.

4 IMPLEMENTATION

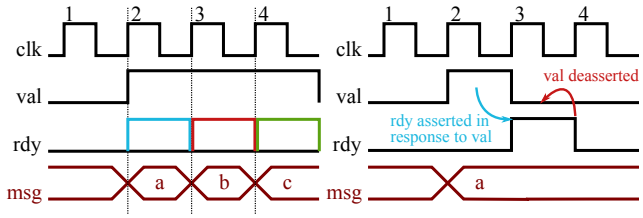
In this section, we describe our implementation of the bounded latency equivalence checking technique in JasperGold, a state-of-the-art commercial formal property verification tool. Section 4.1 describes the specifications of the key properties in the verification harness in the SystemVerilog Assertion language. Section 4.2 discusses how we improve JasperGold's performance by incorporating proof acceleration modules into the verification harness.

To provide an overview of our BLEC implementation, Figure 4 shows an example workflow with our implementation of BLEC: we implement a Verilog parser that assumes the naming of ports in a latency-insensitive interface, which generates the verification harness and JasperGold configuration scripts using a templated approach; the JasperGold FPV tool either finds a counterexample to the stall invariant property (in which case the designers can debug the potential design issues with a waveform from the counterexample) or proves the stall invariant property with respect to the bounded FIFO size (in which case the verification engineer can stop or increase the FIFO size for higher confidence of the proof).

4.1 Property Specification in SystemVerilog Assertion

In this section, we discuss how to specify some of the critical assumptions and properties in the SystemVerilog Assertion (SVA) language [18]. These are assumptions and properties are embedded in the verification harness and are generated by the Verilog parser in a templated fashion. While solving the formal property verification problem, JasperGold will assume the constraints to be true and try to find counterexamples to the asserted properties.

Constraints on Toplevel LI Interface – As discussed in Section 3.2, each ingress latency-insensitive interface in the target DUV will add a toplevel LI interface which streams messages to the ingress interface in the strict and perturbed DUV. However, an unconstrained LI interface of three ports (`val`, `rdy`, and `msg`) may not implement the correct LI handshake behaviors. Figure 5 shows two possible bugs when each of the three ports are allowed to change independently from each other. Figure 5 (a) shows a bug where the downstream module may accept a wrong message because `msg` is allowed to change while `val` is asserted. Figure 5 (b) demonstrates a bug where the downstream module tries to acknowledge a non-existent transaction because `val` gets deasserted before a previous `val` is acknowledged.



(a) Wrong Message (b) Phantom Message

Figure 5: Bugs in Unconstrained Latency-Insensitive Interface – (a) `msg` may change while `val` is asserted; the downstream module may sample a wrong message depending on when `rdy` is asserted. (b) `val` may get deasserted before a previously asserted `val` is acknowledged by `rdy`; if `rdy` is asserted in response to `val`, the downstream module may end up acknowledging an non-existent transaction.

To ensure correct LI handshakes, we add the following assumption to the toplevel LI interface to constrain its behavior.

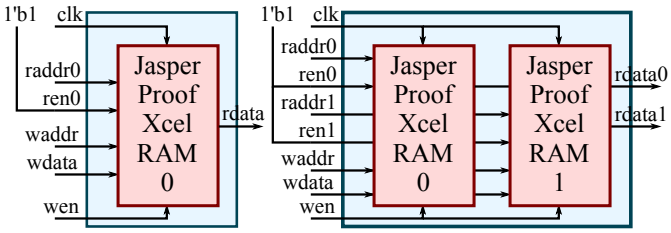
```
li_ifc_asms: assume property (
  @(posedge clk) disable iff (reset) (
    (val |-> rdy) or
    (val |=> ($stable(msg) & $stable(val))
              s_until_with (val & rdy))
  )
);
```

In the above assumption, `|->` and `|=>` are implication operators in the SVA language that indicates the *consequent* (right hand side of the operator) is true if the *antecedent* (left hand side of the operator) is true [1]. The difference between `|->` and `|=>` is that `|->` requires the consequent to be true *at the same cycle* when the antecedent is true; `|=>` requires the consequent to be true *at the next cycle* after the antecedent becomes true. This assumption uses the `s_until_with` operator, which indicates that `msg` and `val` have to remain stable *at the same cycle* `val & rdy` becomes true. This assumption states that at any non-reset cycle, if `val` is asserted, then either `val` and `rdy` are asserted at the same cycle or `val` and `msg` remain stable until the transaction is acknowledged (`val & rdy`).

Properties of Equivalence Checkers – As mentioned in Section 3.1, the equivalence checker checks if the results from the strict and perturbed paths are the same when both egress FIFOs are not empty. We formalize this equivalence check into the following SVA assertion, which guards the check with an antecedent of both `val` signals asserted.

```
same_msg_ast: assert property (
  @(posedge clk) disable iff (reset) (
    (s_val & p_val) |-> (s_msg == p_msg)
  )
);
```

However, the `same_msg_ast` assertion alone is not sufficient to capture all violations to the stall invariant property. Consider one category of violations where the perturbed DUV fails to assert the `val` signal on the egress interface at all. In this case, the formal property verification tool considers this property to be *vacuously true* because the antecedent of the `same_msg_ast` assertion is false [1]. To detect this category of design bugs, we add the following SVA assertion.



(a) Register File 1r1w (b) Register File 2r1w

Figure 6: Register Files with Integrated Proof Acceleration RAM – 1r1w: one read port and one write port; 2r1w: two read ports and one write port.

```
same_vals_ast: assert property (
  @(posedge clk) disable iff (reset) (
    (s_val & ~p_val) |->
      s_eventually (s_val & p_val)
    and
    (~s_val & p_val) |->
      s_eventually (s_val & p_val)
  )
);
```

The assertion `same_vals_ast` has the same consequent among its two clauses which indicates that `s_val & p_val` will become true in some future cycle. The `s_eventually` operator provides a way to express that some event will happen after a finite but uncertain number of cycles. This assertion indicates that no matter which DUV (strict or perturbed) asserts the egress `val` signal, the other DUV will eventually assert its `val` as well.

4.2 Proof Acceleration

To reduce the run time of the verification tool, our BLEC implementation incorporates JasperGold's proof acceleration modules into commonly used RTL modules. Proof acceleration modules are behavioral modules that have built-in behaviors in JasperGold and can be verified more efficiently than their manually implemented RTL counterparts. We specifically target the RTL RAM modules because (1) they generally contain a large number of states and the increasing number of states often strongly correlates with longer tool run time [1]; (2) the RTL RAM modules are widely reused across IPs including FIFOs, register files, caches, and behavioral memories.

Figure 6 shows how we integrate the RAM proof acceleration module into two kinds of register files. For the register file with one read port and one write port (1r1w), we wrap the proof acceleration RAM within the regular register file module and connect all ports accordingly. The read enable port on the proof acceleration RAM is driven by high voltage because the register file is read every cycle. For the register file with two read ports and one write port (2r1w), we duplicate the proof acceleration RAM within the register file module to support simultaneous reads. The write address and data are applied on both proof acceleration RAM. Since we do not modify the interface of the register files, our integration of proof acceleration modules reduces the run time of verification without changing the RTL code of the DUV.

Design	Flip-Flops	Gates	RTL Lines	FIFO Depth
PE	113	729	143	2
GCD	66	655	490	2
Proc.	5983	86830	4898	2

Table 1: RTL Modules and BLEC Parameters Used in Case Studies – PE: the latency-insensitive processing element; GCD: the greatest common divisor unit; Proc.: the RISC-V processor

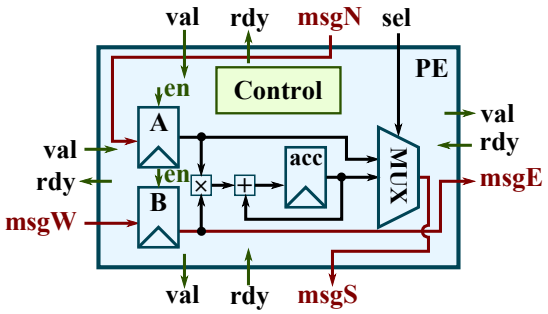


Figure 7: Latency-Insensitive PE – N, W: ingress interface on the north and west side of PE; E, S: egress interface on the east and south side of PE. acc: accumulation register.

5 CASE STUDIES

In this section, we perform case studies on the following three RTL modules with our implementation of BLEC to demonstrate its effectiveness in detecting numerous design bugs: a latency-insensitive processing element (PE), a greatest common divisor (GCD) unit, and a RISC-V processor. We use JasperGold FPV 2023.03 as our formal verification tool and run our case studies on a commodity server with 72 cores of Intel Xeon E7-8867 v4 CPU and 256 GiB of main memory. Table 1 shows the number of flip-flops, gates, the lines of RTL code, and the BLEC parameters used in our case studies.

5.1 The Latency-Insensitive Processing Element

The first case study is on a latency-insensitive processing element (PE) RTL module which is intended to be used as sub-modules of a latency-insensitive systolic array. Figure 7 shows the architecture of the PE module. The PE takes input from two LI interfaces at the north and west side and produces output to the east and south LI interfaces. The PE also performs multiply-accumulation and stores the sum into its internal accumulation register. The PE also forwards the west message to the east side. Depending on the selection input signal, the PE either forwards the north message or the accumulation result to the south side.

Bug: Incorrect Ingress Ready Condition – We examine a PE bug discovered from the commit history of an in-house systolic array (performing matrix multiplication) git repository. According to the commit history, the designers created wrong control logic for the ingress `rdy` signals: `rdy` from the east egress interface was simply bypassed to the west ingress interface and `rdy` from the south egress interface was bypassed to the north ingress interface. This bug created an incorrect ingress ready condition (ingress ready should be true only if both `rdy` from the east and the south side interface are true) which escaped the designer's unit test because the behavioral downstream module of PE always applies egress stalls at the same cycle.

Our implementation of BLEC detects this bug in under ten seconds. JasperGold finds a 5-cycle counterexample to the `same_msg_ast` assertion in the equivalence checker: the strict DUV in the counterexample registers `msgN` and `msgW` at the same cycle; the perturbed DUV has one cycle of egress stall on the east interface, which causes `msgW` to be registered one cycle later than `msgN`. This difference in the timing of registering ingress messages eventually leads to different results from the strict path and the perturbed path.

The PE designer initially identified this bug with a manually crafted test case which captures the exact timing of egress stalls required to trigger this bug. With the waveform of this counterexample derived from BLEC, the PE designer is able to identify and fix the root cause of the failed assertion much faster without manipulating the timings of egress stalls.

Bounded Proof: PE is Stall Invariant – After fixing the ingress ready condition bug, we also leverage BLEC to generate a bounded proof that the PE module is stall invariant. We observe that JasperGold is not able to converge on the PE design because the single-cycle multiplier (two 32-bit inputs, one 32-bit output) in the PE datapath significantly increases the complexity of verification. To help the FPV tool converge, we leverage the fact that the precise multiplier functionality is not required in BLEC. Therefore, we can replace the complex multiplier logic with a much simpler bit-wise XOR operation to improve converge time. Since the LI handshake logic does not depend on the multiply-accumulate result, performing this replacement does not affect the equivalence properties BLEC tries to prove. After replacing the single-cycle multiplier with bit-wise XOR gates, JasperGold is able to prove both the `same_msg_ast` assertion and the `same_vals_ast` assertion within 1.5 hours.

5.2 The Greatest Common Divisor Unit

Our second case study design is a greatest common divisor (GCD) unit which computes the GCD of two input 32-bit integers using a subtraction-based Euclidean algorithm. Figure 8 shows the RTL GCD unit and the finite state machine (FSM) in its control unit. The GCD unit has one ingress LI interface to stream in the two input integers within a single bundle and one egress LI interface to stream out the result. In this case study, we examine and detect two bugs with our BLEC implementation and also prove that the correct GCD unit is stall invariant.

Bug: Unconditional Transition from CALC to DONE – The first bug we investigate is when the control FSM transits unconditionally from the DONE state to the IDLE state. With this bug, the GCD unit may not send out the result correctly if the downstream module is not ready in the cycle GCD unit is in the DONE state. However, this bug is only observed if there is more than one cycle of stalls on the egress interface, which helps the bug escape some simulation-based testing that assumes no egress stalls on the DUV.

Our implementation of BLEC detects this bug in under one minute. JasperGold finds a 7-cycle counterexample to the `same_msg_ast` assertion in the equivalence checker: the toplevel LI interface generates two messages into the two instances of DUV; the egress

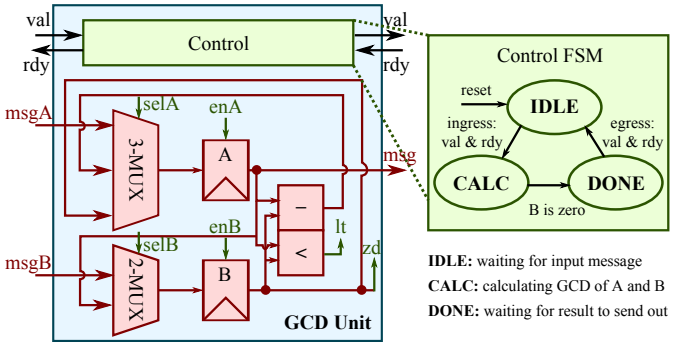


Figure 8: GCD Unit – lt: if A is less than B; zd: if B is zero.

perturber applies one cycle of stall on the egress interface, which causes the first result of the perturbed DUV to drop; the equivalence checker therefore finds the first result from the strict DUV and the second result from the perturbed DUV to be different, triggering a failed assertion. Verification engineers can deduce from the counterexample waveform that the DUV has different behaviors under different stall conditions, which helps debugging.

Bug: Wrong Transition Condition from CALC to DONE – The second bug creates a wrong transition condition where the FSM only transits to DONE if the egress interface is ready and transits to IDLE otherwise. With this bug, the GCD unit will function correctly if there is no egress stalls; but the DUV will not generate valid output messages if there is egress stalls. Similar to the unconditional transition bug, this bug can escape simple simulation tests that assume no egress stalls.

BLEC detects this bug in under one minute. JasperGold identifies that the `same_msgs_ast` assertion vacuously passes (i.e., the antecedent condition is unreachable) because under this bug the strict and the perturbed DUV cannot generate a valid output message at the same cycle (perturbed DUV has at least one cycle egress stall). But JasperGold does find a counterexample of infinite length to the `same_vals_ast` assertion: the toplevel LI interface generates two input messages and the strict DUV produces two output messages before becoming idle; the perturbed DUV does not generate any output and remains idle for the rest of the trace. Similar to the unconditional transition bug, verification engineers can leverage the counterexample to debug the design issue.

Bounded Proof: GCD Unit is Stall Invariant – We also leverage BLEC to generate a bounded proof that the GCD unit without bugs is stall invariant. We make two minor changes to the GCD unit design to help the FPV tool converge without undermining the stall invariant proof.

First, we make the observation that for large 32-bit inputs, the GCD unit may spend a significant number of cycles in the `CALC` state to compute the greatest common divisor using the subtraction-based Euclidean algorithm. Therefore, formally verifying the complete 32-bit GCD unit design is intractable because the FPV tool has to examine all 32-bit input pairs and step through the Euclidean algorithm calculation to find potential violations of the stall invariant property. To help the FPV tool converge on the GCD unit design, we modify the state transition condition from state `CALC` to `DONE` to expedite the GCD computation process. As shown in Figure 8, the control FSM in the GCD unit transits from `CALC` to `DONE` when the registered B value is zero. We remove this condition and make the transition to the `DONE` state unconditional. This change effectively reduces the number of cycles required to compute the greatest common divisor.

Second, we apply a similar change to the bitwise-XOR operation in the latency-insensitive PE to avoid reasoning about complex computations in the GCD unit datapath. As shown in Figure 8, the datapath of the GCD unit includes a subtraction operation between the registered A and B values. We replace the subtraction operation with a bitwise-AND operation so that the FPV tool can reason about simpler bitwise-AND operations instead of a 32-bit subtraction.

Both of the above changes do not undermine the stall invariant proof because the changes only affect logic outside of the GCD unit's handshake control logic. After applying the above two changes, JasperGold is able to prove both the `same_msg_ast` assertion and the `same_vals_ast` assertion in the GCD verification harness within 20 minutes.

5.3 The Pipelined RISC-V Processor

Our final case study design is a five-stage pipelined RISC-V processor that implements the RV32IM instruction set [3]. Figure 9 shows the simplified datapath and control diagram of the pipelined processor used in this case study. The target processor RTL module communicates to the instruction memory and data memory through four memory interfaces: memory requests are transferred through the `imem_req` and `dmem_req` interfaces, and memory responses come back through the `imem_resp` and `dmem_resp` interfaces. Internally, the processor has five pipeline stages: fetch (F), decode (D), execution (X), memory (M), and write-back (W). The processor reads the register file at stage D and writes back to the register file at stage W. The processor has a simple branch predictor that always predicts not taken. In the event of a branch mis-prediction (`jmp` or `br_taken`), the processor squashes stage F (if a jump instruction) or stage F and D (if a branch instruction) to discard invalid states. Each pipeline stage may also *originate a stall* (`ostall` signals) in the event of hazards or when memory responses have not arrived, which stalls all stages after the originating stage.

The BLEC verification harness of the processor is different from that of the previous case studies. We make the observation that the memory request latency-insensitive interfaces of the processor are inherently stall variant: branch instructions may squash earlier memory requests and therefore memory response stalls can lead to different informative memory requests. We choose to implement equivalence checking on the RISC-V verification interface [24], which exposes the states of the processor in instruction commit order and is guaranteed to be stall invariant regardless of instruction and data memory stalls. The right side of Figure 9 shows some of the exposed processor states used in our case study. `val` is the latency-insensitive valid signal which indicates if the output signals are valid at a cycle; `order` is a counter that keeps track of the number of committed instructions; `insn` is the 32-bit instruction; `pc_rdata` is the PC register value for the current instruction and `pc_wdata` is the PC register value for the immediate next instruction; `x_wb` is a bit vector that tracks which architectural register is written;

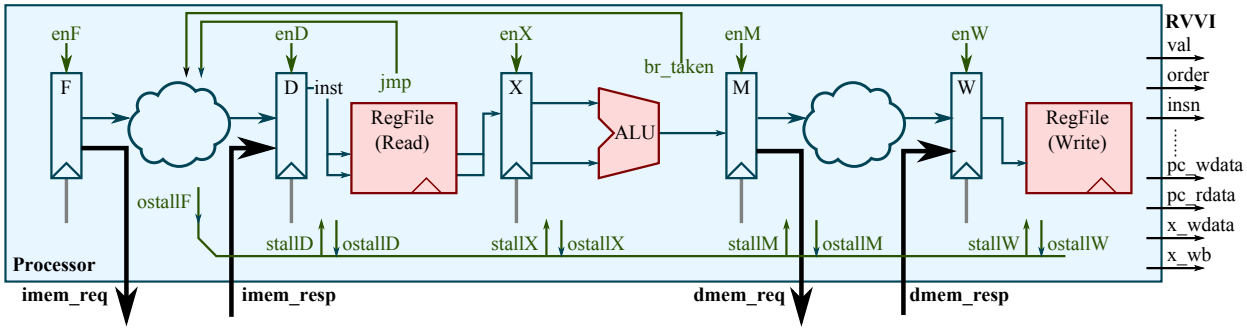


Figure 9: Pipelined Processor – RVVI: RISC-V verification interface. `ostall`: if this pipeline stage is originating an event that stalls (`stall`) this stage and all stages after; `jmp`: squash stage F if the decoded instruction is a jump instruction; `br_taken`: squash stage F and D if the current instruction is a branch and the branch is taken. Each thick black arrow represents one latency-insensitive val-rdy interface.

	Addr.	Order	18	19	20	21	22	23	24	25
Strict	0x20	7	F	D	X	M	W			
	0x24	8		F	D	X	M	W		
	0x28	9				F	D	X	M	W
Pert.	0x20	7	F	D	X	M	W			
	0x24	8			F	D	X	M	W	
	0x28	9				F	D	X	M	W

Table 2: Counterexample of Stall Invariant Property in Processor Bug – Perturb.: perturbed DUV. Instructions before order 7 setup the architectural registers and are omitted here. All three instructions shown here are branch instructions that are taken. Stages in blue generate different egress informative events on the processor's `imem_req` interface in the strict DUV and the perturbed DUV. Stages in red are the root cause (solid arrow) to the difference. The dashed arrow indicates the causal relationship in the correct processor.

`x_wdata` is the content of all architectural registers at instruction commit.

To provide instruction and data memory responses, we also include a behavioral memory backed by the JasperGold proof accelerator described in Section 4.2. To retain generality of our method, we do not fill the behavioral memory and instead allow the FPV tool to the memory response message as free variables. To reduce the FPV tool time, we also add the assumption to the processor decode stage that all instructions at the decode stage is a valid RISC-V instruction.

Bug: Not Squashing F/D When Branch is Taken – We examine a bug where the processor does not correctly squash stage F and D when a mis-predication happens. For the processor in Figure 9, this bug is equivalent to clamping the `br_taken` signal in stage X to low, which does not discard the invalid instruction at stage D.

JasperGold finds a 27-cycle counterexample to the `same_msg_ast` property on the instruction memory request interface within 30 minutes. This counterexample includes 10 valid RISC-V instructions, 5 of which are integer arithmetic instructions that setup the architectural register values and the other 5 branch instructions trigger the bug. Table 2 shows the pipeline diagram of the last three instructions (all three branches are taken) in the bugged processor (both strict DUV and perturbed DUV). Blue pipeline stages generate the different messages (i.e., different PC register values) on the instruction memory request interface (i.e., the fetch stage at cycle 21). In the strict DUV, the X stage at cycle 20 should have squashed the F stage and fill the PC register with the correct branch target address. However, at cycle 20 the pipeline stalls, and the existence of the squash bug eliminates the supposed address update. Instead, the PC register (stage F) at cycle 21 holds the immediate next address after the PC register value at cycle 19. In the perturbed DUV, the pipeline continues to progress at cycle 20, and the PC register at cycle 21 holds the updated branch target address.

Attempted Bounded Proof: Processor is Stall Invariant – Despite being able to find violations of the processor's stall invariant property within a relatively short period of time, in our case study JasperGold cannot establish a proof of the equivalence properties in BLEC verification harness within a reasonable amount of time (48 hours wall time). The main reason for the extended time to converge is the processor register file and the instruction and data memory. The target RISC-V processor includes a register file of 32 32-bit entries, and the instruction and data memory both have 64 32-bit entires (we choose a small number of memory entries to reduce converge time). These RTL memory modules represent an enormous state space, which the FPV tool has to exhaustively search through to eventually generate a bounded proof of the stall invariant property.

We have attempted several methods to reduce the processor complexity by introducing extra constraints. For example, we add assumptions that certain RISC-V instructions will not appear to reduce the decoder complexity; we remove the support for several arithmetic operations in the ALU; we also reduce the bitwidth of the long data bus (`x_wdata`) in the RVVI to shrink the state space the FPV tool needs to search through. Future research may need to further reduce the state space of the verification harness to eventually establish a bounded proof of the stall invariant property.

5.4 Discussions

Based on our experiences performing the above case studies, we observe that BLEC is effective at detecting bugs in the given latency-insensitive RTL modules. Our FPV tool (JasperGold) usually takes a reasonably short period of time to discover a counterexample to the

stall invariant property in the original RTL module. As a concrete example, in the RISC-V processor case study, JasperGold discovers a counterexample of 27 cycles in the original processor RTL with uninitialized behavioral instruction and data memory in under 20 minutes.

However, it usually takes the FPV tools significantly longer time to achieve a bounded proof of stall invariant on the given RTL module, and some manual changes are necessary to help the FPV tool converge faster. Fortunately, BLEC is compatible with many design changes that can significantly reduce tool converge time. Most of these changes reduce the complexity of the target DUV's datapath by replacing complex computations (typically with a large number of gates) with simpler computations. Since the latency-insensitive handshake logic of most DUVs do not depend on the exact values of these computations, those changes generally do not undermine the stall invariant proof. Concrete examples of those changes include replacing the multiplication logic with bitwise-XOR gates (PE case study) and replacing the subtraction logic with bitwise-AND gates (GCD unit case study).

6 RELATED WORK

Bounded model checking [12] is a formal verification technique which verifies if a given transition system obeys the specification of its intended behaviors. The industry has adopted bounded model checking based formal verification techniques to verify the functional correctness of large RTL designs [2, 6, 7, 13]. Both these existing works and our work leverage bounded model checking based formal verification methods to prove or find counterexamples to the intended behaviors of RTL modules. However, there are two major differences between the above existing works and our work. First, existing works mainly focus on verifying the functional correctness of the RTL modules and our work focuses on finding violations of the stall invariant property. Second, to achieve a detailed and unambiguous specification, the above existing works mainly reply on manual specifications of intended behaviors of an RTL module. This requires intimate knowledge of both the design's functionalities and the specification language, which limits formal methods' accessibility to a relatively small audience. In contrast, our proposal democratizes the formal verification techniques by encapsulating details of the specification into verification modules (perturbers and equivalence checkers).

Carloni et al. propose a correct-by-construction methodology to develop latency-insensitive designs using a helper modules including channels, relay stations, and shells [9]. *Shells* are wrapper modules around the target DUV to enable correct-by-construction latency-insensitive communications with other LI channels. The authors claim that a shell can be automatically synthesized from a given DUV, which reduces the time required to implement a correct latency-insensitive RTL module. In the face of stalling events, the shell stalls the wrapped DUV instance through clock gating to preserve its internal states and only allows state changes when all input messages have become valid. Comparing to our work, Carloni et al.'s proposal represents an orthogonal correct-by-construction solution to the verification challenge of latency-insensitive designs.

Researchers have also explored properties similar to the stall invariant property and applied it in other contexts. Dai et al. propose to leverage formal verification techniques to validate high-level synthesis (HLS) results based on the latency-equivalence of the design under different inputs [14]. Piccolboni et al. propose to formally verify the latency equivalence of different high-level synthesis results to achieve high confidence in HLS results. Piccolboni's proposal, KAIROS, assumes an incremental modification workflow and verifies if the result of each synthesis step produces results that are latency equivalent to the reference module. Similar to our proposal, Dai et al. and Piccolboni et al. also construct a verification harness with latency-insensitive input manipulation logic. However, both our work and their proposals have different focuses and represent orthogonal efforts on tackling HLS verification issues and a more traditional ASIC/FPGA prototyping verification challenges. Suhaib et al. propose to validate LI components by verifying the latency-equivalence between a LI component and its synchronous counterpart, both of which are described using a verification modeling language [21]. Our work focuses on verifying the stall invariant property of LI components modeled at RTL, which includes most of the hardware modules used in ASIC and FPGA prototyping. Wijayasekara investigates a similar property to the stall invariant property in the context of asynchronous circuits and tackles the verification challenges in the asynchronous context [25], where as our work focuses on the correctness of digital LI components.

7 CONCLUSIONS

Despite its success in enabling hardware standard libraries and numerous network-on-chip IPs, latency-insensitive protocols have imposed a unique verification challenges on RTL modules where existing simulation-based dynamic verification techniques require significant efforts to build test suites that cover a large number of stall conditions. In this paper, we propose a formal verification methodology to address the verification challenge of latency-insensitive RTL modules. We introduce the stall invariant property of latency-insensitive RTL modules and make the observation that most bugs in LI modules are violations of the stall invariant property. We propose bounded latency equivalence checking, which constructs a verification harness accepted by a formal property verification tool to find inconsistent latency-insensitive behaviors under different stall conditions. We implement our proposed BLEC technique with a state-of-the-art commercial formal verification tool and perform three case studies to evaluate its effectiveness. Our case studies demonstrate that BLEC can find all injected bugs within relatively short period of time. The case studies also find existing commercial formal verification tools can provide a bounded proof of the stall invariant property on many manually simplified RTL modules.

ACKNOWLEDGMENTS

We thank Marcelo Orenes Vera (Princeton University) and Dr. Cunxi Yu (University of Utah) for their insightful comments at an early stage of this work. We thank Dr. Shunning Jiang (Cornell University) for contributing to the processor implementation. We also thank the anonymous reviewers for their helpful comments on the manuscript. This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, a research gift from Xilinx, Inc., and donations from Intel.

REFERENCES

[1] 2015. *SVA: The Power of Assertions in SystemVerilog*. Springer International Publishing.

[2] David P. Appenzeller and Andreas Kuehlmann. 1995. Formal Verification of a PowerPC Microprocessor. *Int'l Conf. on Computer Design (ICCD)* (1995).

[3] Krste Asanovic and David Patterson. 2014. *Instruction Sets Should Be Free: The Case for RISC-V*. Technical Report UCB/EECS-2014-146. EECS Department, University of California, Berkeley.

[4] Luca Benini and Giovanni De Micheli. 2002. Networks on Chips: A New SoC Paradigm. *IEEE Computer* (2002).

[5] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages Twelve Years Later. *Proc. of the IEEE* (2003).

[6] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. 1999. Verifying Safety Properties of a PowerPC-Microprocessor Using Symbolic Model Checking without BDDs. *Int'l Conf. on Computer-Aided Verification (CAV)* (1999).

[7] Per Bjesse, Tim Leonard, and Abdel Mokkedem. 2001. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. *Int'l Conf. on Computer-Aided Verification (CAV)* (2001).

[8] Luca P. Carloni. 2015. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proc. of the IEEE* (2015).

[9] Luca P. Carloni, Kenneth L. McMillan, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. 1999. A Methodology for Correct-by-Construction Latency Insensitive Design. *ICCAD* (1999).

[10] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 1999. Latency Insensitive Protocols. *Int'l Conf. on Computer-Aided Verification (CAV)* (1999).

[11] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 2001. Theory of Latency-Insensitive Design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (Sep 2001).

[12] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design* (2001).

[13] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. 2001. Benefits of Bounded Model Checking at an Industrial Setting. *Int'l Conf. on Computer-Aided Verification (CAV)* (2001).

[14] Steve Dai, Alicia Klinefelter, Haoxing Ren, Rangharajan Venkatesan, Ben Keller, Nathaniel Pinckney, and Brucek Khailany. 2021. Verifying High-Level Latency-Insensitive Designs with Formal Model Checking. *cs.LO arXiv:2102.06326* (Feb 2021).

[15] William J. Dally and Brian Towles. 2001. Route Packets, Not Wires: On-Chip Interconnection Networks. *DAC* (2001).

[16] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. 2012. Leveraging Latency-Insensitivity to Ease Multiple FPGA Design. *FPGA* (2012).

[17] Thomas A. Henzinger and Joseph Sifakis. 2007. The Discipline of Embedded Systems Design. *IEEE Computer* (2007).

[18] IEEE. 2017. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. Online Webpage. https://ieeexplore.ieee.org/document/8299595.

[19] Cheng-Hong Li, Sampada Sonalkar, and Luca P. Carloni. 2010. Exploiting Local Logic Structures to Optimize Multi-Core SoC Floorplanning. *DAC* (2010).

[20] ARM Ltd. 2011. AMBA AXI and ACE Protocol Specification. (2011).

[21] Syed Suhaib, Deepak Mathaikutty, David Berner, and Sandeep Shukla. 2006. Validating Families of Latency Insensitive Protocols. *IEEE Trans. on Computers (TC)* 55, 11 (Nov 2006), 1391–1401.

[22] Cheng Tan, Yanghui Ou, Shunning Jiang, Peitian Pan, Christopher Torng, Shady Agwa, and Christopher Batten. 2019. PyOCN: A Unified Framework for Modeling, Testing, and Evaluating On-Chip Networks. *Int'l Conf. on Computer Design (ICCD)* (2019).

[23] Michael Bedford Taylor. 2018. Basejump STL: SystemVerilog Needs a Standard Template Library for Hardware Design. *DAC* (2018).

[24] RISC-V Verification. 2023. RISC-V Verification Interface. Online Webpage. https://github.com/riscv-verification/RVVI.

[25] Vidura Manu Wijayasekara. 2016. *Equivalence Verification for NULL Convention Logic and Latency-Insensitive Circuits*. Ph.D. Dissertation. Department of Electrical and Computer Engineering, North Dakota State University.