2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Quiescent Formal Checks (QFC) for Detecting Deep Design Bugs – Sooner and Faster

Somesh Mishra, Mayank, Kumar, Ketan Mishra, Anshul Jain, Bharath Varma Gottumukkala
Intel Corporation
{somesh.mishra, mayank.kumar, ketan.mishra, anshul.jain, bharath.varma.gottumukkala}@intel.com

***Abstract*- Quiesce Formal Checks (QFC) is a novel approach for pre-silicon formal verification of RTL implementation of control designs. QFC relies on the bounded model checking (BMC) technique for finding deep functional bugs in the designs without requiring extensive design-specific properties or a full formal design specification. Formal verification is a well-established technique for efficient and exhaustive verification of complex designs; however, its results are often limited by state-space explosion and proof complexity on large industrial designs. QFC aims to increase the effectiveness of formal verification by marrying the "Quiesce State" concept from the software applications world with formal technology as a bug-hunting strategy. In this paper, we present the work on QFC, its applications, and results demonstrating its practicality and effectiveness in various designs.**

I. Introduction

Formal verification is a technique used to ensure the correctness of complex designs, such as hardware systems or software applications, by mathematically proving that the design behaves as intended. This method is widely recognized as a powerful tool for detecting errors in designs, but it faces certain challenges when applied to large industrial designs. Two of the main limitations are state-space explosion and proof complexity, which can make the verification process time-consuming and resource-intensive.

Quiesce Formal Checks (QFC) is a new approach to address these challenges, specifically for pre-silicon formal verification of RTL (Register Transfer Level) implementations of control designs. The main idea behind QFC is to combine the "Quiesce State" concept, which comes from the software applications domain, with formal verification techniques. This combination serves as a bug-hunting strategy that aims to improve the effectiveness of formal verification.

In the context of QFC, this concept is used to identify specific points in the design where verification can be focused, thus reducing the complexity of the problem. QFC relies on the bounded model checking (BMC) technique, which is a method for finding deep functional bugs in designs without the need for extensive design-specific properties or a complete formal design specification. BMC works by exploring the state space of a system within a given bound, making it more efficient than traditional model checking techniques.

In order to demonstrate the practicality and effectiveness of QFC, we present various applications and results. They show that QFC can be successfully applied to different designs and can help detect bugs that would otherwise be difficult to find using traditional formal verification methods.

II. Related Work

A. Concept of Quiesce State in Software Applications

The Quiesce State in software applications refers to a temporary inactive state in which a system, program, thread, or database is paused or halted. This concept is widely used in computer science to manage various aspects of software systems. A quiesce command is issued to force all users to pause their activities, effectively putting the system (e.g., network link, database, etc.) into "quiesce mode." This allows system administrators to gain exclusive access to the system for performing administrative tasks without interference from ongoing user activities. By placing the system into a Quiesce State, administrators can carry out critical tasks such as system maintenance, upgrades, debugging, or data backup, without the risk of corrupting the data or causing conflicts with other operations. This approach ensures consistency and stability of the system during the execution of these tasks. Commonly followed process for quiescing systems in software applications is shown in figure 1.

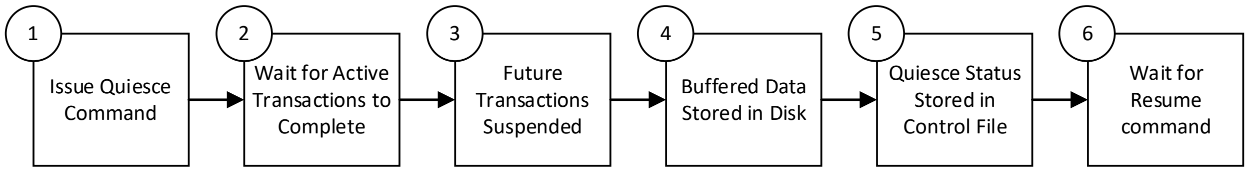
1
Issue Quiesce Command
2
Wait for Active Transactions to Complete
3
Future Transactions Suspended
4
Buffered Data Stored in Disk
5
Quiesce Status Stored in Control File
6
Wait for Resume command

Figure 1: Commonly Used Quiescing Process

In the context of QFC, the concept of Quiesce State is applied to the verification of RTL (Register Transfer Level) implementations of control designs. The idea is to inject a "quiesce command" into the design, which brings it to a halt and allows for the probing of interesting internal signals within the design. This approach enables an effective method for checking functional consistency of the design.

B. Concept of Floating Pulse in Formal Verification

The concept of floating pulse in formal verification refers to a technique that allows formal tools to assert a single pulse at an arbitrary time after a reset, during the course of a formal trace. This pulse represents a special event, typically used to mark a particular transaction, packet, or other significant occurrence within the system. The purpose of tracking these special events is to facilitate the comparison of the system's behavior against a formal reference model, which is essential for verifying the correctness of the design. In the context of formal verification, the floating pulse is strategically placed by the tool to expose potential design flaws, ultimately producing a counterexample that demonstrates the issue. The timing of the pulse is not predetermined but is instead chosen by the tool based on its ability to reveal weaknesses in the design. A working example of floating pulse in context of this paper is shown in figure 2.

```
reg pulse_seen;
wire pulse;
always @(posedge clk) begin
   if (rst) pulse_seen <= 1'b0;
   else if (pulse)
      pulse_seen <= 1'b1;
end

pulse_model: assume property (
   @(posedge clk) disable iff (rst)
   pulse_seen |-> !pulse
);
```

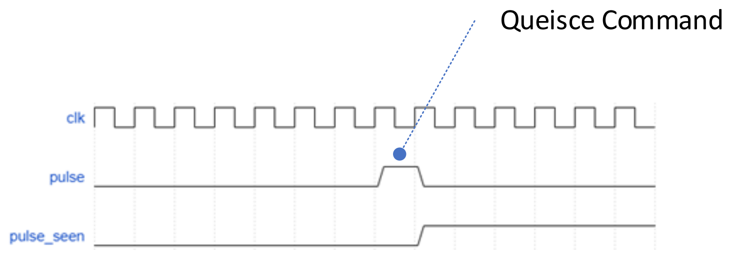

Figure 2: Floating Pulse Implementation

QFC utilizes the floating pulse concept to create a sense of "quiesce command" for the design under test. By introducing a floating pulse, QFC effectively halts the design at an arbitrary point, enabling the verification process to focus on specific aspects of the design and probe internal signals for consistency. This approach helps to identify potential design flaws, improve the effectiveness of formal verification, and overcome some of the challenges faced by traditional formal verification methods, such as state-space explosion and proof complexity.

III. PROBLEM STATEMENT

A. Bug Activation-Detection Gap

Sophisticated bugs in designs, such as hardware or software systems, often exhibit a common behavior where they do not immediately cause issues but rather surface after a certain delay or "warm-up" period. This means that the bug becomes activated only after specific events or conditions have occurred within the system. The activation of such bugs may not lead to immediate detrimental effects, but over time, they can cause significant problems in the design that can be detected by high-level end-to-end checkers.

As illustrated in Figure 3, we can consider a hypothetical scenario where a design has two events, event A at cycle 3 and event B at cycle 5. These events can be considered as the "warm-up" period for the bug, during which it becomes activated. However, it is not until cycle 10 that the bug's activation leads to issues within the design.

Once the bug becomes activated, it may still take some time before it causes noticeable detrimental effects in the system. This delay can make it challenging to identify and address these sophisticated bugs, as they may not manifest immediately and can be difficult to trace back to their root cause. High-level end-to-end checkers are used to detect these types of issues, as they monitor the overall behavior of the system to identify inconsistencies or errors.

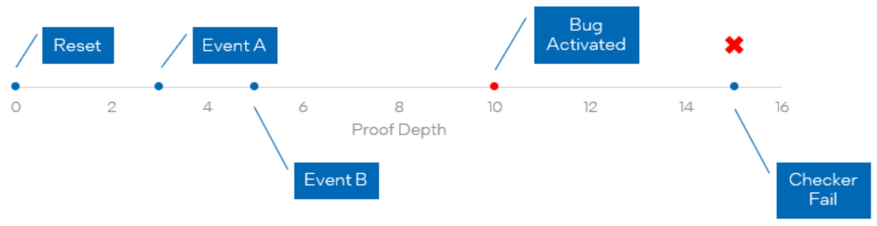

Figure 3: Bug Activation-Detection Gap

B. *Formal Complexity*

Formal technology is a powerful method for finding design bugs due to its ability to perform an exhaustive, breadth-first search of the state space of a given system. By systematically exploring all possible states and transitions, formal tools can effectively identify design flaws that may be difficult to detect using other techniques. However, this same attribute introduces challenges such as state-space explosion, which can make it difficult for formal tools to reach deeper states of the design.

State-space explosion occurs when the number of possible states and transitions in the design grows exponentially, making it increasingly difficult for formal tools to explore the entire state space within a reasonable amount of time and resources. This issue can be exacerbated by factors such as the size of the design (number of gates, flops, latches) and the cone-of-influence (COI) of end-to-end checkers, which can further increase the complexity of verification.

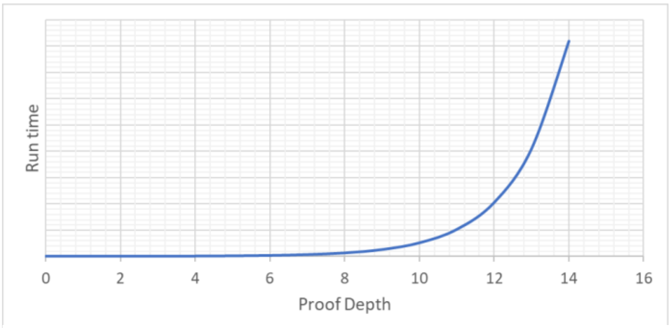

Figure 4: Exponential Complexity

As a result, formal tools may reach a point where they are unable to continue exploring the state space effectively, giving up before reaching the depth where a checker would fail and potentially miss critical design flaws. Figure 4 (not provided) illustrates the effect of state-space explosion and formal complexity by showing how the formal tool's effort and time required to reach higher proof depths increase exponentially.

C. *Undetected Bugs*

The combination of bug activation-detection gap and formal complexity can result in undetected bugs, even when a capable formal verification environment is set up. This is because the delay between the activation of a bug and its detection may outpace the checker's proof depth, leading to missed bugs.

Figure 5 illustrates how the delay in bug activation and detection may outpace the checker's proof depth, resulting in missed bugs. In this scenario, the formal tool may not be able to explore deep enough into the state space to detect the bug, as the bug's activation and detection occur beyond the tool's proof depth.

Developing strategies and techniques to address these challenges is crucial to improving the effectiveness of formal verification in detecting complex design bugs.

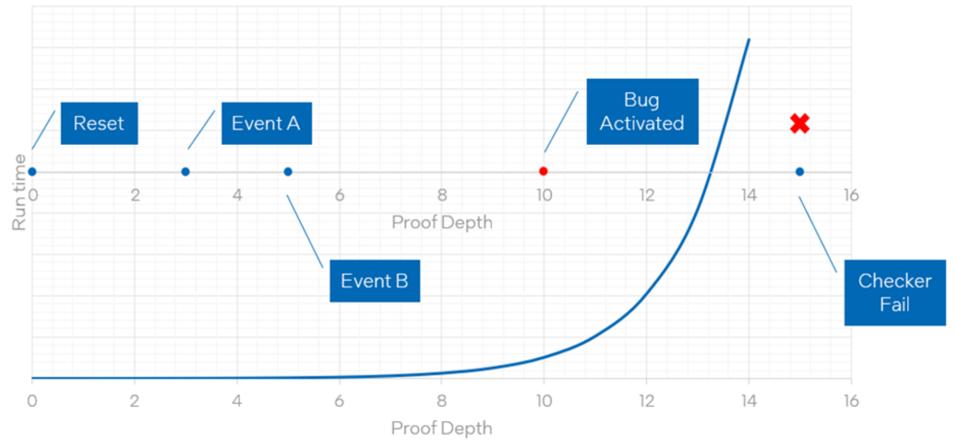

Figure 5: Bugs Beyond Exponential Complexity Wall

IV. Quiesce Formal Checkers (QFC)

A. *Methodology*

Quiesce Formal Checks (QFC) aims to address the challenges posed by the bug activation-detection gap and formal complexity by providing a framework through which effective checkers can be implemented on important signals of the design. These checkers monitor both internal and output signals to quickly identify and flag bugs as soon as they are activated. Figure 6 captures the overall process to implement QFC for design under test.

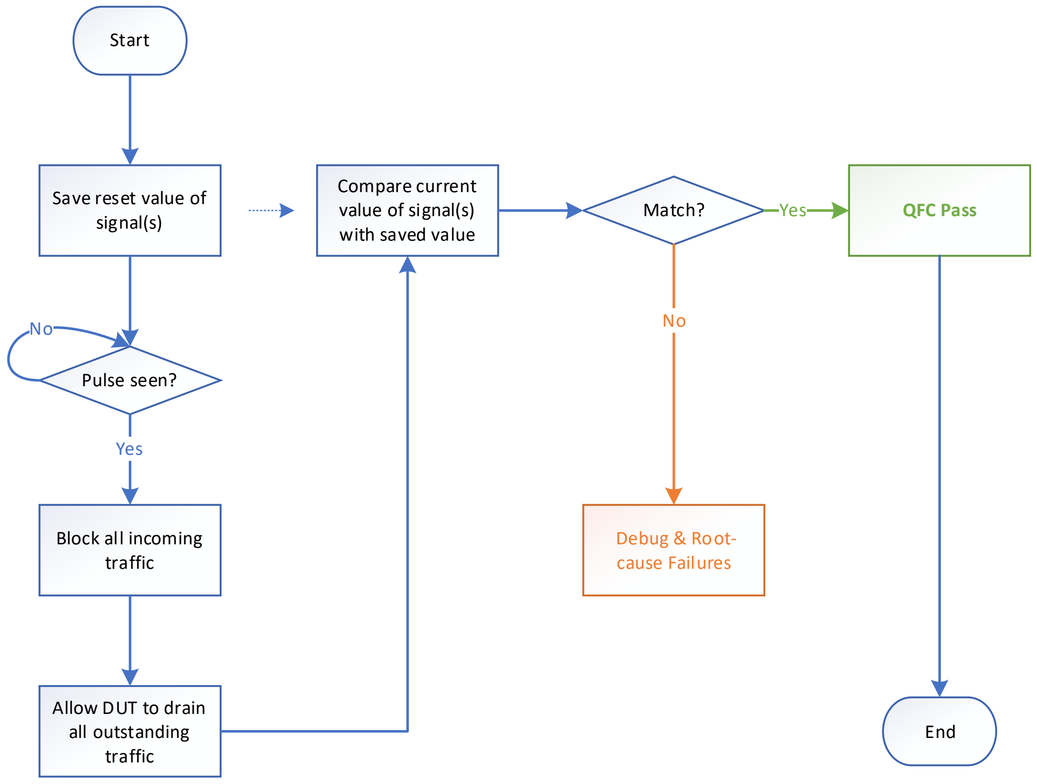
Start
Save reset value of signal(s)
Pulse seen?
No
Yes
Block all incoming traffic
Allow DUT to drain all outstanding traffic
Compare current value of signal(s) with saved value
Match?
Yes
No
QFC Pass
Debug & Root-cause Failures
End

Figure 6: QFC Methodology

QFC methodology, as outlined in Figure 6, typically involves the following steps:

#	*Step*	*Description*
1	Identify key signals	The first step in the QFC framework is to identify critical internal signals and output signals within the design that are most likely to be affected by bugs or contribute to their activation.
2	Implement checkers	Checkers are implemented to monitor these important signals. These checkers are designed to flag inconsistencies as soon as bugs are activated, minimizing the gap between bug activation and detection.
3	Inject quiesce commands	To improve the effectiveness of the checkers, the QFC framework employs quiesce commands and floating pulses. Quiesce commands temporarily halt the design, allowing for the probing of internal signals to ensure consistency. Floating pulses, on the other hand, are strategically placed by the tool to expose potential design flaws, leading to counterexamples that demonstrate the issue.
4	Run formal verification	With the checkers in place, the formal verification process is carried out. The QFC framework enables the formal tool to effectively explore the state space while minimizing the impact of state-space explosion and formal complexity.
5	Analyze results	Finally, the results of the formal verification are analyzed. If bugs are detected, the checkers can be refined to improve their effectiveness in identifying and flagging bugs as they are activated.

B. Use-case

Resource leakage is a functional issue that frequently occurs in control designs that manage resources like buffers, linked-lists, FIFOs, and similar structures. When implementation mistakes are made in these designs, resources can be lost in corner-case scenarios, resulting in the design losing one resource each time a leak takes place. Typically, these scenarios involve the design failing to properly capture the released resources.

Control designs are responsible for managing resources, ensuring their correct allocation, and releasing them when they are no longer needed. When a resource is not correctly captured or released due to an implementation error, it can lead to resource leakage. In corner-case scenarios, these errors can become more apparent, and the control design loses resources gradually.

Resource leakage can have several adverse effects on the system, including performance degradation, starvation of transactions, and potential deadlocks. It is crucial to identify and address these resource leakage issues early in the design process to ensure that the system operates efficiently and effectively. Let us look into the adverse effects of resource leakage and how can these effects be mitigated using QFC.

1. *Performance degradation*: When resources are leaked due to implementation mistakes, the design's capacity, and bandwidth to handle incoming traffic are negatively affected. As a result, the overall performance of the design declines over time. Each leaked resource represents a loss of available resources for the design to manage incoming traffic efficiently. This can lead to several issues, including increased latency, reduced throughput, and slower response times for processing requests. As more resources are leaked, the control design becomes less capable of handling incoming traffic effectively, causing the performance of the system to degrade further. To prevent performance degradation due to resource leakage, QFC can be implemented to ensure that number of design resources replenishes to its original value after finite duration of quiesce operation.
2. *Starvation*: If an active resource is leaked, it means that a particular transaction will be forgotten by the design forever, causing that transaction to starve. This situation can have severe implications, especially when other transactions in the system depend on the completion of the starved transaction. When a transaction is starved, it is unable to make progress towards its completion, potentially blocking or delaying other transactions that rely on its successful completion. If enough transactions are affected, it can result in a cascading effect, causing the entire system to slow down or even come to a complete halt after a long gap since forgetting the first transaction. To prevent starvation caused by resource leakage, QFC can be implemented to ensure no traces of transactions are left in the design after finite duration of quiesce operation.
3. *Deadlock*: If all the resources are leaked gradually over time, the design becomes unable to function, causing the system to deadlock. A deadlock occurs when two or more processes in a system are stuck, waiting for each other to release a resource, and none of them can proceed further. In the case of resource leakage, as resources

are lost over time, the design eventually runs out of available resources to handle incoming traffic or process transactions. When this happens, the system can no longer make progress, leading to a deadlock situation. To prevent deadlocks caused by resource leakage, QFC can be implemented to ensure all pending transactions have been flushed out within finite duration of quiesce operation.

V. Implementation Details

The following source code defines a SystemVerilog module called **`simple_qfc`** that serves as a checker for testing the behavior of a design under test (DUT) in response to a quiesce command. The quiesce command is used to bring the DUT to a stable state where no new transactions are processed, and all pending transactions are completed.

The module takes several parameters:

1. **`MAX_PENDING_TXNS`**: The maximum number of pending transactions that the DUT can support
2. **`MAX_WAIT_TIME`**: The maximum time within which the DUT should return to a quiescent state
3. **`SIG_WIDTH`**: The width of the signal being checked

The module has three main inputs (apart from clock and reset signals):

1. **`incoming_txn_vld`**: A signal that indicates a new incoming transaction is valid
2. **`outgoing_txn_vld`**: A signal that indicates an outgoing transaction is valid
3. **`sig_under_check`**: The signal being checked for its behavior during quiesce

There are two main assertions in this module:

1. **`check_quiesce_consistency`**: This asserts that the signal under check returns to its out-of-reset value within the specified MAX_WAIT_TIME after the quiesce command is seen and there are no pending transactions.
2. **`check_quiesce_stability`**: This asserts that the signal under check acquires a stable value within the specified MAX_WAIT_TIME after the quiesce command is seen and there are no pending transactions.

Additionally, there is an assumption **`input_stop_at_quiesce_indication`** that ensures that no new incoming transactions are allowed after the quiesce command has been seen.

```
module simple_qfc #(
   // maximum number of transaction that the DUT can support
   parameter MAX_PENDING_TXNS = 1,
   // max drain time within which DUT should return to quiesce state
   parameter MAX_WAIT_TIME = 1,
   // width of signal being checked
   parameter SIG_WIDTH = 1
) (
   // inputs to the checker
   input logic clk, reset,
   input logic incoming_txn_vld,
   input logic outgoing_txn_vld,
   input logic [SIG_WIDTH-1:0] sig_under_check
);

reg reset_del;
always @(posedge clk) reset_del <= reset;

// store out-of-reset value of the signal under check
reg [SIG_WIDTH-1:0] oor_val;
always @(posedge clk) begin
   if(reset_del) oor_val <= sig_under_check;
end

// model insertion of quiesce command at random design state using floating pulse method
wire quiesce_pulse;
…
…
…

// counter to track pending transaction in the DUT
localparam PENDING_CNTR_WIDTH = $clog2(MAX_PENDING_TXNS + 1);
logic [PENDING_CNTR_WIDTH-1:0] pending_txns;
…
…
…

// checker to ensure that signal under check returns to its out-of-reset value
// within finite duration of quiesce command
check_quiesce_consistency: assert property (
   @(posedge clk) disable iff (reset)
   (pending_txns == 'd0) && quiesce_pulse_seen |->
   ## MAX_WAIT_TIME (sig_under_check == oor_val)
);

// checker to ensure that signal under check acquires a stable value within
// finite duration of quiesce command
check_quiesce_stability: assert property (
   @(posedge clk) disable iff (reset)
   (pending_txns == 'd0) && quiesce_pulse_seen |->
   ## MAX_WAIT_TIME (sig_under_check == $past(sig_under_check))
);

// constraint to stop incoming transaction after quiesce command is seen
…
…
…

endmodule
```

This module can be used with formal verification tools to test the behavior of a design under test (DUT) using instantiation example shown below.

```
simple_qfc #(
   .MAX_PENDING_TXNS (…),
   .MAX_WAIT_TIME (…),
   .SIG_WIDTH (…)
) inst_qfc (
   .clk    (…),
   .reset  (…),
   .incoming_txn_vld (…),
   .outgoing_txn_vld (…),
   .signal_under_check (…)
);
```

VI. CONCLUSIONS

In conclusion, Quiesce Formal Checks (QFC) is an innovative approach that aims to improve the efficiency and effectiveness of formal verification for complex designs. By combining the Quiesce State concept with formal technology and relying on bounded model checking, QFC offers a promising bug-hunting strategy that can overcome some of the limitations faced by conventional formal verification techniques. Further research and development in this area could lead to even more powerful and practical verification methods for large-scale industrial designs.

ACKNOWLEDGMENT

Thanks to the design team and FVCTO Team for the motivation and the valuable feedback.

REFERENCES

[1] IBM, “Quiesce command”. IBM Documentation, https://www.ibm.com/docs/en/db2/11.5?topic=commands-quiesce, Jan 2023.
[2] IBM, “Unquiesce command”. IBM Documentation, https://www.ibm.com/docs/en/db2/11.5?topic=commands-unquiesce, Jan 2023.
[3] IEEE Std 1800™-2017, IEEE Standard of System Verilog – Unified Hardware Design, Specification, and Verification Language.
[4] M, Achutha KiranKumar, Erik Seligman & Tom Schubert, Book on “Formal Verification – An Essential Toolkit for VLSI Design”, 2015.
[5] Mark Moir; Nir Shavit (2007). "Concurrent Data Structures". In Dinesh Metha; Sartaj Sahni (eds.). Handbook of Data Structures and Applications. Chapman and Hall/CRC Press. pp. 47-14–47-30