Effective Pre-Silicon Verification of Processor Cores by Breaking the Bounds of Symbolic Quick Error Detection

Abstract

Existing techniques to ensure functional correctness or detect hardware Trojans in processor cores during pre-silicon verification face severe limitations. We extend Symbolic Quick Error Detection, a recent bounded model checking-based bug detection technique, with symbolic starting states. Using open-source RISC-V cores, we demonstrate: 1. Quick (≤5 minutes (≤2.5 hours) for an in-order (superscalar) core) detection of 100% of hundreds of logic bug and hardware Trojan scenarios from commercial SoCs and research literature, and 97.9% of "extremal" bugs (randomly-generated bugs requiring ~100,000 activation instructions taken from random test programs). 2. Quick (~1 minute) detection of several previously unknown bugs in open-source RISC-V designs.

1. Introduction

Pre-silicon verification requires major effort in a typical design flow [Foster 15]. This paper focuses on pre-silicon verification of single processor cores, which are critical components of any System-on-Chip (SoC). Generally, pre-silicon verification mainly targets logic design errors (*logic bugs*). Beyond logic bugs, future pre-silicon verification techniques might also need to address Hardware Trojans (*HTs*) [King 08], which are unauthorized modifications of an integrated circuit (*IC*), resulting in incorrect functionality and/or the exposure of sensitive data [Karri 10]. While initial research on HTs focused on attacks implemented at the foundry during fabrication, there are also concerns about HTs being inserted in third-party Intellectual Property (*IP*) cores by malicious sources [Zhang 11]. Processor cores are an important class of IP cores that are used (and reused) across multiple designs to reduce development costs and meet aggressive time-to-market requirements. Thus, ensuring functional correctness and trustworthiness of processor cores is an important problem.

Similar to logic bugs, we consider HTs which affect functionality; i.e., such an HT can cause an error that can (ultimately) create an error in the software-visible state (software-visible registers or memory). This encompasses many demonstrated attacks on processor cores [King 08].

Symbolic Quick Error Detection (QED) [Singh 18] is a new pre-silicon verification technique based on ideas from QED [Hong 10, Lin 12, 14, 15a]. It uses Bounded Model Checking (BMC) [Clarke 01] for formal analysis of the design. It is an automated and fast bug detection and localization technique. This contrasts with traditional pre-silicon verification techniques that are mostly ad-hoc and manual, and that fail to scale (background is given in Appendix B).

Symbolic QED analyzes a design symbolically, but it requires a concrete (non-symbolic, i.e., with 0's and 1's specified explicitly) starting state. Starting from a concrete state means that, to find bugs that require very long activation sequences (i.e., many instructions are required to activate such bugs), Symbolic QED must rely on very deep BMC runs (i.e., runs that unroll the circuit far enough to include all the activation instructions). This may be difficult for practical designs. For example, [Singh 18] states that a BMC tool can only unroll a design for up to around 30 clock cycles, within 24 hours of verification time. The following HT example, taken from [Zhang 14], shows that Symbolic QED, while highly effective for many logic bugs, can be insufficient for detecting HTs.

Motivating Example

Consider the following HT which may be difficult (see Sec. 5 for more discussion) to find during pre-silicon verification using existing HT detection techniques [Salmani 13, Zhang 14]: *The HT changes the opcodes of the next several decoded instructions after it sees a specific sequence of 256 instructions fetched.*

This HT could inject a short sequence of instructions to bypass physical memory protection and run a privileged instruction. This is an example of a privilege escalation attack [King 08], that can be catastrophic.

Because the HT requires many instructions (and many clock cycles) for activation, Symbolic QED (like other BMC-based methods [Rajendran 15, Reece 16]) fails to detect the example, unless the starting state quickly transitions to the state where the HT activates. Trying to stumble upon such a "close" state by picking a concrete state from simulation or starting from a reset state is highly unlikely to succeed, since the HT could be designed with an arbitrary activation sequence that is not known *a priori*.

To overcome this major challenge, we extend Symbolic QED so that it is now capable of starting from a symbolic (instead of concrete) starting state (i.e., we give the BMC tool the ability to choose an arbitrary starting state). As explained in the next paragraph, BMC from a symbolic starting state can be highly challenging. Our approach overcomes the associated challenges and allows us to start BMC-based Symbolic QED from states that would otherwise require very deep BMC runs that aren't supported by current formal tools. As a result, even logic bugs that require very long activation sequences can be detected, by applying a very short sequence of instructions after starting from a "close" state. Of course, we can also detect the motivating HT example within minutes as well.

Existing BMC-based pre-silicon verification techniques (including existing Symbolic QED) face the following challenge when used in conjunction with symbolic starting states: BMC needs a property to check along with property-specific constraints (assumptions made by the BMC tool about the design) on the symbolic starting state. These constraints are necessary to prevent the BMC tool from producing spurious counter-examples (false positives) for the property it checks. If the BMC tool selects a starting state that is not reachable from the set of all valid reset states of the system using a valid sequence of instructions, a false positive might occur. For example, assume that each word in a memory is protected with a single even parity bit. Assume that a BMC tool is asked to check the following property: for any sequence of reads and writes to the memory, the parity bits remain consistent with the data. If the starting state of the design is not constrained, the BMC tool can initialize the memory to contain an all-zero word with a '1' for the parity bit, read from this memory location, and report this spurious counterexample (which is a false positive). Traditional techniques therefore rely on the verification engineer to manually write constraints to rule out such false positives, which can be extremely time-consuming for large designs with many complex properties.

This paper overcomes the above challenge by: i) defining *QED constraints*: sufficient constraints to ensure false positives do not occur when using Symbolic QED with symbolic starting states in a single processor core; and ii) introducing *QED recorders,* which observe a small subset of internal signals within the processor, and make sure the QED constraints are satisfied. QED recorders are used in pre-silicon verification only. They do not incur area overhead for the final design.

The following are some of the key experimental results that we report:

1) We automatically, correctly, and quickly (within 1 minute) detected several real logic bugs that were previously unknown in an open-source superscalar RISC-V core [Ridecore1].
2) We automatically, correctly, and quickly (within 25 seconds for an in-order core; 18 minutes for a superscalar core) detected 100% of hundreds of simulated logic bugs, representing a wide variety of "difficult" scenarios (see Appendix A) that occurred in various commercial designs.
3) We automatically, correctly, and quickly (within 5 minutes for an in-order core; 2 hours for a superscalar core) detected 100% of hundreds of simulated Hardware Trojans, representing a wide variety of scenarios (see Appendix A) from research literature. In contrast, Symbolic QED with a concrete starting state detected only 9% of these HTs.
4) We automatically, correctly, and quickly (within 2.5 hours) detected 97.9% of a family of "extremal" bugs (randomly-generated pre-condition-based bugs which require ~100,000 activation instructions taken from random test programs) in a superscalar core. In contrast, Symbolic QED with a concrete starting state detected 0% of these bugs.

This work builds on previous work on Symbolic QED with symbolic initial states [Fadiheh 18]. More discussion is provided in Appendix B.3.

The following are some of the important features of our technique

1) It is highly effective for detecting both logic bugs and HTs (despite long activation sequences) during pre-silicon verification of in-order and superscalar cores, as demonstrated by our results.

2) It does not require the verification engineer to manually craft design-specific assertions to find logic bugs or HTs.
3) No false positives, as demonstrated by our results.
4) It does not require a "golden" model or simulation data of the design under test for detection of logic bugs and/or HTs.
5) Its effectiveness does not depend on the way HTs are designed, i.e., our method is HT-design agnostic.

The rest of the paper is organized as follows. Sec. 2 describes how we extend Symbolic QED with symbolic starting states. Results are presented in Sec. 3, followed by related work in Sec. 4. Sec. 5 concludes.

Appendix A presents a classification of bugs and HTs. Appendix B provides background and a survey of earlier QED works, and introduces a new *QED module* (a small block used only during pre-silicon verification of the design, that allows the BMC tool to pass Symbolic QED instructions to the processor core), used in our experiments in this paper. Appendix C of [Extended 18] explains our assumptions about how bug-free designs operate, and Appendix D of [Extended 18] provides details on how the QED constraints are specified to the formal tool.

2. Extending Symbolic QED with Symbolic Starting States

In this section, we discuss how Symbolic QED [Singh 18] is extended with symbolic starting states (background on QED and Symbolic QED is given in Appendix B). As stated in the introduction, the main challenge with symbolic starting states is that of false positives: the BMC tool might select a starting state that is not reachable from the set of all valid reset states using a valid sequence of events (instructions) that could lead to a false positive. To avoid false positives, we define a set of constraints (called *QED constraints*) on the symbolic starting state (see Sec. 2.1). To implement these QED constraints, we introduce *QED recorders*, additional hardware modules (used only during pre-silicon verification—they do not contribute area to the final design) that record a small subset of internal logic values of the processor core and check that the QED constraints are satisfied (see Sec 2.2). The inputs and steps compared with Symbolic QED without symbolic starting states is shown in Fig. 1.

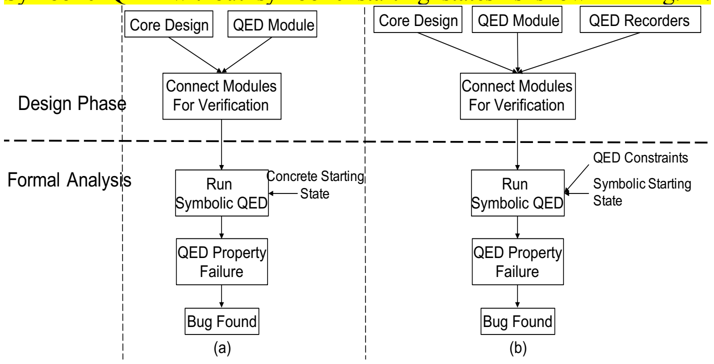

Fig. 1. (a) Symbolic QED inputs and steps without symbolic starting states, and (b) Symbolic QED inputs and steps with symbolic starting states.

2.1. QED Constraints

We first define some terminology which we use in the constraint definitions: i) *Symbolic In-Flight (SIF) "instructions"*: symbols (i.e., state bits), chosen by the BMC tool as part of the symbolic starting state, corresponding to values stored in the (microarchitectural) flip-flops within the pipeline that hold instructions during normal operation. During normal operation, these flip-flops are populated by an instruction passing through the core[1]; ii) *Symbolic QED instructions*: symbols which represent the instructions that form the bug trace (which is part of the counter-example, along with the starting state that BMC assigns) generated by the BMC tool; iii) T_C: the point in time when all SIF instructions commit (i.e., write to architectural state). It is determined by the BMC tool; and iv) *Symbolic QED operand data:* symbols representing the operand[2] data (an operand can be either an architectural register or a memory location) of dispatched Symbolic QED instructions (dispatched before T_C)

Fig. 2 illustrates these definitions for a three-stage in-order pipeline. When the formal analysis begins, there are up to three SIF instructions, and all commit by time T_C. The first Symbolic QED instruction (*R1=R1+5*) is fetched into the pipeline, and its Symbolic QED operand data is available after it passes through the Dispatch stage.

Now, the QED constraints are given as follows (see Appendix D of [Extended 18] for more details on how each constraint is enforced):

Constraint C-1: At T_C, All SIF instructions have committed (i.e., no SIF instruction can write to the architectural state after T_C), while all Symbolic QED instructions commit after T_C.

Constraint C-2: At T_C, the architectural state is QED consistent (see Appendix B for definition). Further, nothing but a Symbolic QED instruction can write to architectural state after T_C (e.g., any test mode such as scan that can directly write to registers must be disabled).

Constraint C-3: All the operand data for each Symbolic QED instruction I, must have one of the following properties:

i) if the operand data is available (i.e., I has already read data for this operand) then this data must match the corresponding register/memory location (i.e., source operand location) data at T_C.

ii)[3] if the operand data is not available at T_C, then I is waiting for the result of an earlier Symbolic QED instruction for this operand data.

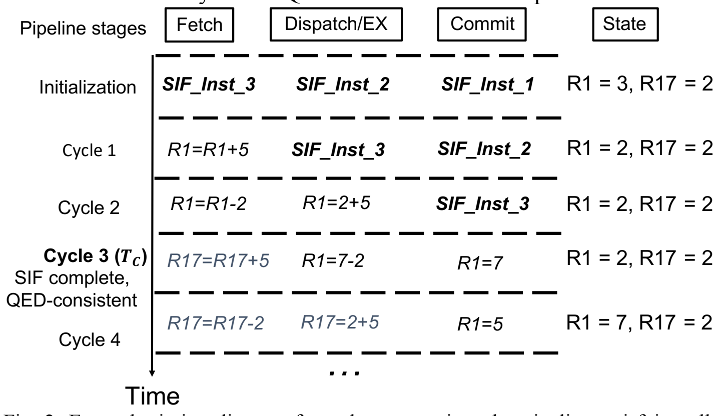

Fig. 2. Example timing diagram for a three-stage in-order pipeline satisfying all QED constraints. SIF instructions commit by Tc, before all QED instructions.

The QED constraints form a sufficient condition to ensure no false positives, given that any bug-free design satisfies two assumptions after T_C. These two assumptions are detailed and justified in Appendix C of [Extended 18]. Details on specifying QED constraints to the formal tool are given in Appendix D of [Extended 18].

Importantly, these same QED constraints do not prevent false positives for general property checking using BMC. For example, let a processor core start at a state where the Exception Program Counter (*EPC*) is misaligned, the current PC is within an exception handling routine, and there are only NOP instructions in the pipeline. This is an unreachable state for processors with strict alignment rules (e.g., MIPS). It makes sense to check the property that the EPC is aligned, since returning to a misaligned address can cause a fault. Even at Tc, when the NOP sequence is finished, this EPC is still be misaligned, causing a false positive.

When using the constraints with Symbolic QED though, we do not get such a false positive. This is because the exception handling routine will be filled by valid QED tests, and any time we assert a QED check, it should not fail unless there is a bug in the design. Even if the core returns to the misaligned address, the processor may fault, but no QED check fails.

2.2. QED Recorders

Here we discuss QED recorders, which are used to record semantic information (about T_C and Symbolic QED operands) so that we can specify the QED constraints to the BMC tool (explained in Appendix D of [Extended 18]). For ease of understanding, we take a core with single instruction fetch, and a 5-stage in-order pipeline as a running example in Sec. 2.2 and Appendix D of [Extended 18], but we also explain how the technique was extended to a superscalar core. In Sec. 3, we present results for both an in-order core and a superscalar core, emphasizing that the technique is not limited to only a certain type of core.

2.2.1. Recorder for T_C

As T_C depends on the starting state chosen by the BMC tool, it cannot be statically determined before the formal analysis begins. So, we add a recorder to the design which can give this information to the formal tool dynamically (Symbolically). For an in-order core, T_C can be determined by simply tracking the progress of the first Symbolic QED instruction until it reaches the commit stage (write-back stage) of the pipeline. At this point in time, we can be sure that all SIF instructions have been committed, as the pipeline is completely occupied by Symbolic QED instructions.

Specifics of the T_C recorder for a 5-stage, single-fetch, in-order pipeline is given in Fig. 3. Inputs are ready signals for all stages that precede the commit stage (e.g., *fetch_ready* is high when the fetch stage is ready to receive an instruction). The output *SIF_complete* is true when the first Symbolic QED instruction goes through all pipeline stages and reaches the commit stage. The output *mode* keeps track of progress made so far by the Symbolic QED instruction (we later make use of this output in Sec. 2.2.2). Note that this T_C recorder for a 5-stage pipeline can be easily modified to support in-order pipelines with an arbitrary number of stages.

For a superscalar core, the T_C recorder is even simpler, and utilizes the reorder buffer (ROB). The basic idea here is to mark the entry allocated in the ROB for the first Symbolic QED instruction. After this, *SIF_complete* is made true when the head pointer of ROB reaches the marked instruction.

```
INPUT: fetch_ready, dispatch_ready, exec_ready, mem_ready
OUTPUT: SIF_complete, mode
// initialization
mode ← S0;  SIF_complete ← false;
// end initialization
if (mode == S0) && (fetch_ready) then
  mode ← S1;  SIF_complete ← false;  // instruction completes fetch stage
end if
if (mode == S1) && (dispatch_ready) then
  mode ← S2;  SIF_complete ← false;  // instruction completes decode stage
end if
if (mode == S2) && (exec_ready) then
  mode ← S3;  SIF_complete ← false;  // instruction completes execute stage
end if
if (mode == S3) && (mem_ready) then
  mode ← S4;  SIF_complete ← true;  // instruction completes memory stage
end if
```

Fig. 3. Pseudocode for T_C recorder.

2.2.2. Recorder for Symbolic QED Operands

Like T_C, we cannot statically determine Symbolic QED operands, as they depend on the starting state. In a Symbolic QED instruction, an operand can either be a register or a memory location. The Symbolic QED operand recorder stores information for both cases. Specifics of the Symbolic QED operand recorder for a 5-stage, single-fetch, in-order pipeline is given in Fig. 4.

```
INPUT: src1_valid, src1_addr, src1_data, src2_valid, src2_addr, src2_data, mem_valid,
mem_addr, mem_data, mode
OUTPUT: src1_buffer, src2_buffer, mem_buffer
// initialization
src1_buffer ← empty_buffer;  src2_buffer ← empty_buffer;
mem_buffer ← empty_buffer;
// end initialization
if (mode != S0) && (mode != S1) && (src1_valid || src2_valid) then
  if (src1_valid) then
    src1_buffer.add_entry(src1_addr, src1_data);
  end if
  if (src2_valid) then
    src2_buffer.add_entry(src2_addr, src2_data);
  end if
end if
if ((mode == S3) || (mode == S4)) && (mem_valid) then
  mem_buffer.add_entry(mem_addr, mem_data);
end if
```

Fig. 4. Pseudocode for Symbolic QED operand recorder.

Inputs are: 1) *_valid*, which is high when *addr* and *data* lines are valid; 2) *_addr*, which gives register/memory address of an operand; 3) *_data*, which gives operand data; 4) *mode*, which gives the state of the T_C recorder, as given in Fig. 3. Output *_buffer* simply gives all Symbolic QED operands and their respective values. Initially all buffers are empty. As we only care about Symbolic QED operands, we only store the information for these operands in buffers i.e., we do not store operand information of any SIF instruction. This is made sure by conditioning, the addition of new entries to buffers, on state of T_C recorder i.e., *mode* (e.g., we do not add new entries to *src1_buffer* until all SIF instructions pass through dispatch stage). Note that in Fig. 4, we assume that instructions can only read at most two register values and one memory location per cycle, but the idea can easily be extended to an arbitrary number of reads.

For a superscalar core, the idea given in Fig. 5 needs to be extended to include Symbolic QED operands that are waiting on results of earlier Symbolic QED instructions. This is done by tagging each operand as either waiting or not inside the buffers. For a waiting operand, we also store the instruction tag (ROB entry number) of the instruction it is waiting for. This waiting information is needed to specify Constraint C-3 for a superscalar core. Further details are given in Appendix D of [Extended 18].

3. Results

We demonstrate the effectiveness of Symbolic QED with symbolic starting states on two open-source RISC-V processor cores. The two processor cores are: i) V-scale [Vscale], an in-order core designed for embedded applications; and ii) RIDECORE [Ridecore1], a superscalar core (2-way pipeline, 64 maximum instructions in-flight, 2 ALUs, 1 multiplier, 1 load/store unit) designed for high performance applications. For BMC, we used the Questa Formal tool (version 10.5c) from Mentor Graphics on an AMD Opteron 6438 with 128GB of RAM. For each core, we instrumented the QED module (given in Appendix B.4) along with appropriate QED recorders in the design and specified the QED constraints as discussed in Sec. 2.1 and 2.2. We simulated "long" logic bugs, which are harder (i.e., requiring very long activation sequences) versions of "difficult" logic bug scenarios that occurred in various commercial SoCs [Singh 18]. These scenarios are given in Appendix A.

Each bug is modeled as having two parts: i) activation criteria of the bug (Table A.1.a), i.e., the conditions which need to be satisfied for the bug to activate; and ii) effect of the bug once it is activated (Table A.1.b). A bug scenario is formed by pairing one bug activation criterion (Table A.1.a) with one bug effect (Table A.1.b). For our experiments, we considered a whole range of values for the parameters in Table A.1, as follows, N=Y={2,4,6,16,32,64,128,256}, R=X={2,4,6,…,30}. This results in a total of 117 logic bugs in Vscale (Activation criterion A.1.a.5 is not possible with Vscale), and 120 logic bugs in RIDECORE. In Table 1, we present our findings.

Table 1. Results for injected "long" logic bugs and HTs. For bug traces, we report the [minimum, average, maximum] length in instructions and clock cycles. We also report [minimum, average, maximum] BMC runtimes (time taken by BMC tool to find a counter-example).

		"Long" Bugs	HTs
Vscale	Bug trace length (instructions)	[2, 2, 3]	[2, 2, 3]
	Bug trace length (clock cycles)	[5, 5, 6]	[5, 5, 6]
	Coverage	100%	100%
	BMC runtime (seconds)	[2, 4, 25]	[2, 11, 313]
RIDE CORE	Bug trace length (instructions)	[4, 4, 4]	[4, 4, 4]
	Bug trace length (clock cycles)	[8, 8, 8]	[8, 8, 8]
	Coverage	100%	100%
	BMC runtime (minutes)	[7, 13, 18]	[7, 20, 121]

Observation 1: Symbolic QED with symbolic starting states correctly and automatically found all "long" logic bugs, in less than 30 mins, without needing to write design-specific assertions or debugging false positives. It found bug scenarios with long activation sequences where traditional BMC methods fail due to small BMC proof bounds.

We also simulated a variety of Hardware Trojan scenarios found in the literature. These scenarios are given in Table A.2. Table A.2.a gives HT activation scenarios, which are chosen to capture all the different HT triggering mechanisms given in the Trust-Hub benchmarks [Salmani 13], while Table A.2.b gives various effects an HT can have on the executing instructions upon activation. Additionally, Table A.2.c, presents the three HT implementation techniques which are commonly used to inject HTs in designs [Rajendran 15]. These techniques are used to design stealthy HTs to evade common detection techniques (e.g., HTs designed using methodology in [Zhang 13a] are known to evade detection techniques based on UCI [Hicks 10] and coverage metrics [Hicks 10, Zhang 11]). A HT scenario is formed by using one activation criteria (Table A.2.a) with one bug effect (Table A.2.b), along with following an appropriate design strategy (Table A.2.c). We used a range of values for the HT scenario parameters, given in Table A.2, $N = \{2, 4, 8, 16, 32, 64, 128, 256\}$, $M_1 = 32$, $X_1 = X_2 = \{128, 256\}$, $M_2 = 64$, resulting in 156 HT scenarios in Vscale (bug effect A.2.b.5, is not possible with Vscale) and 195 HT scenarios in RIDECORE. The parameter values are chosen such that the HT activation is extremely rare and are similar, or higher (i.e., harder to activate), than the values used in the benchmark Trojans of [Salmani 13].

Observation 2: Symbolic QED with Symbolic starting states correctly and automatically found all HTs, in less than 2.5 hours, without needing to write design-specific assertions or debug false positives. It found HT scenarios where state-of-the-art HT detection techniques [Hicks 10, Waksman 13, Zhang 11, Zhang 13b] are ineffective.

In addition to successfully finding all the above bugs and HTs, for RIDECORE, Symbolic QED with symbolic starting states found three previously unknown logic bugs in the design. The first bug was found in under 30 mins, and it was localized to a specific reservation station entry. After fixing this bug and rerunning Symbolic QED with symbolic starting states, we found a second bug in the design, and then a third one after fixing the previously found bug. Both these bugs were also localized to the same reservation station entry in the reservation station (RS-m) of the multiplier unit inside the core. All three discovered bug instances were confirmed by RIDECORE designers [Ridecore2]. Details are in Table 2.

Importantly, these three bugs were not detected by running Symbolic QED as in [Singh 18]. These bugs were detected because of the new QED module introduced in this paper (see Appendix B.4). These bugs require that two multiply instructions (one original and one duplicate) occur in successive clock cycles to observe the bug effect. The QED module of [Singh 18] is not sufficient due to a waiting period required (to achieve a flushed pipeline state) between original and duplicate instructions. In short, our new QED model improves the QED module of [Singh 18] (more details are provided in Appendix B.4) by removing the pipeline flushing, and allowing arbitrary interleaving between original and duplicate instruction subsequences. All three bugs could still be detected with Symbolic QED starting from a power-on reset state, if the new QED module was used (i.e., they did not necessarily require a symbolic starting state). In addition, the runtimes are significantly improved.

Table 2. Found bugs in RIDECORE. We report details on bug activation and effect, along with BMC runtimes in minutes (with symbolic starting states), and seconds (with concrete power-on reset starting state).

Bug Activation	Bug Effect	Runtime (Symbolic; mins)	Runtime (Concrete; secs)
All, but one (the buggy entry) RS-m entries are occupied, and a MULH[6] instruction is assigned to the vacant entry.	First source operand of the MULH instruction corrupted	25	63
All, but one (the buggy entry), RS-m entries are occupied, and a MULH instruction is assigned to the vacant entry.	Second source operand of MULH instruction corrupted	61	69
All, but one (the buggy entry), RS-m entries are occupied, and a MULHU[7] instruction is assigned to the vacant entry.	Result of the MULHU instruction corrupted	64	93

In addition to the bugs in RIDECORE, Symbolic QED confirmed two bugs in Vscale, which were found by running Symbolic QED starting at a concrete reset state in less than 40 seconds. The bugs were first discovered by running S^{2}QED [Fadiheh 18] (see Appendix B.3 for details) and confirmed with the designers. These bugs are due to errors in the Vscale implementation of the RISC-V privileged ISA [RISCVP], within specific Control Status Registers (CSRs). Importantly, Vscale does not implement shadows for CSRs. Thus, when testing these commands, we ensured through the formal tool that a "duplicate" shadow was maintained in data memory for each CSR register (otherwise the condition for *Assumption-1* of Appendix C of [Extended 18] is not satisfied, even in a bug-free design).

The first bug occurs because two bits in the MIP register that cause an interrupt every time they are set high in RISC-V implementations can be overwritten in the Vscale implementation. After this bug was fixed, a second bug was found which occurs when the lower two bits of the MSTATUS register (which govern the privilege level of the design) are set to illegal values. Details are provided in Table 3.

Table 3. Confirmed bugs in VSCALE. We report details on bug activation and effect, and BMC runtimes in seconds. Runtimes are for Symbolic QED with concrete, power-on reset starting state.

Bug Activation	Bug Effect	BMC Runtime
The value '1' is written to specific bit positions in the machine-interrupt CSR MIP.	MTIMECMP register corrupted; Causes repeated interrupts.	2s
Any value with lower two bits set to '01' or '10' written to the machine-level CSR MSTATUS.	Design enters unspecified privilege level; MEPC register corrupted;	33s

We further tested the robustness of our technique on a family of "extremal" bugs that are only triggered when the design reaches a specific set of states. We implemented a bug injection framework that automatically generates these "extremal" bugs by following these steps: i) First it runs a specified test program on the design in simulation and stops the simulation at a random point in time; ii) It runs a uniform random sequence of 100 R-type, I-type, and Load/Store instructions; iii) Selects a uniformly random subset of flip-flops from the set of all flip-flops in the design and dumps their logic values in a file; and iv) Generates a bug (with a pre-defined effect), which can be injected into the design, that is only activated when the design reaches a state where all the selected flip flops in step iii) have the values given by the state dumped in step iii). As the sampling of the flip flops may represent a simulation state which can only be reached by executing many instructions in simulation, the bugs generated are expected to have very long activation sequences.

We implemented the above framework on top of the RIDECORE simulation setup and we used a test program provided with the design which does Matrix Multiply and takes over one million clock cycles to complete (this is a normal test program, not a QED test). Using the framework, we injected logic bugs with different activation criteria into RIDECORE, having the bug effect given in Table A.1.b.3. We present our results in Table 4. For generating these bugs, we randomly chose 180 simulation time points for our experiments, and the number of clock cycles elapsed from the start of the program to each of these simulation time points ranged from 26026 to 988159. For each simulation time point, we generated 10 bugs by randomly selecting 128 flip flops from the simulation dump 10 times, thereby resulting in a total of 1800 bugs.

We were able to find 1763 of the 1800 bugs which were injected into the design, while Symbolic QED starting from a QED consistent reset state was unable to detect even one of these 1800 bugs within 24 hours. In the remaining cases, the BMC tool timed out after 24 hours, before generating the bug trace. In Table 4, we do not consider these bugs as detected.

[1]The formal tool is free to choose any values for Symbols (state bits) associated with in-flight instructions, including those which are not consistent with the logic driving those Symbols. Thus, the values chosen for the symbols in a Symbolic in-flight "instruction" may or may not constitute a valid in-flight instruction.
[2]Operands may come from either registers or memory locations. For register operands, the dispatch stage is the register read stage, while for memory operands the "dispatch" stage is the memory read stage.
[3]This condition is required for out-of-order cores where there is a possibility that the Symbolic QED operand may wait on a SIF instruction instead of a Symbolic QED instruction.

[4]For e.g., if the first instruction in the QED test sequence is allocated ROB entry 10 and a SIF instruction is chosen such that its ROB entry is 15. Note that this peculiar ROB allocation should not happen in any of the reachable states of the processor, but this may happen when we start from a Symbolic state, as we are not constraining the design to only start from reachable states.
[5] MULH is a signed multiply instruction selecting the upper half of the multiplier result.
[6] MULHU is an unsigned multiply instruction selecting the upper half of the multiplier result.

Importantly, Symbolic QED with a concrete starting state as in [Singh 18] could not detect even one of these bugs, regardless of whether the QED module of [Singh 18] or the new QED module of Appendix B.4 was used, the power-on reset state was used as the starting state, or a concrete QED-consistent starting state taken from a random QED test was used.

Table 4. Results for "extremal" logic bugs. For bug traces, we report the [minimum, average, maximum] length in instructions and clock cycles. We also report [minimum, average, maximum] BMC runtimes.

	"Extremal" Bugs
Bug trace length (instructions)	[4, 4, 4]
Bug trace length (clock cycles)	[8, 8, 8]
Coverage	97.9%
BMC runtime (minutes)	[8, 33, 109]

Observation 4: Symbolic QED with symbolic starting states correctly and automatically found most of the simulation state pre-condition-based logic bugs and generated a bug trace in less than 3 hours.

Finally, we compare the bug/HT coverage achieved by using Symbolic QED with a Symbolic starting state vs. Symbolic QED with concrete starting state in Table 5. For all bug/HT scenarios for RIDECORE, we present the efficacy of Symbolic QED starting from a power-on reset state.

Table 5. Comparison between Symbolic QED and Symbolic QED with symbolic starting state in terms of bug coverage for RIDECORE.

Error type	**Error Count**	**Detected by Symbolic QED**	**Detected by Symbolic QED with symbolic starting state**
"Long" logic bugs (Table 1)	120	6	120
HTs (Table 1)	195	17	195
"Extremal" bugs (Table 4)	1800	0	1763
Total	2118	23	2031

Observation 5: Symbolic QED with symbolic starting state achieves a significant improvement in bug coverage in comparison to using Symbolic QED with a concrete starting state.

4. Related Work

Existing pre-silicon verification methods mainly use one or more of the following techniques: i) Software simulation; ii) emulation/hardware acceleration; and iii) formal verification. Software simulation-based techniques are typically extremely slow and require significant manual effort for writing "good" test-benches. Some effort can be reduced by using directed test generation and constrained-random simulation [Adir 04, Bergeron 03, Fine 03, 06, Gutkovich 06, Ioannides 10, Katz 12, Kitchen 07, Mishra 02, Shyam 06, Yuan 99], but the effectiveness of these techniques to find corner case bugs is questionable [Adir 11, Mitra 10]. Hardware acceleration [Boule 05, Chatterjee 12, 13, Kim 04a, 04b, Mammo 12, Mavroidis 07] can be used to reduce simulation time, but still requires manually crafted "good" test-benches to find bugs, and significant effort to port the design to an emulation and checker environment [Chatterjee 13]. The capabilities of tools using symbolic logical analysis for verification have improved dramatically since the seminal works [Burch 92, 94, Clarke 86], and have been used to verify abstracted models of processors [Lahiri 01, 03] and the RTL [Bormann 07, Nguyen 08, Reid 16], by using a set of specifications given as assertions [Foster 03] to the formal tool. However, choosing the "right" set of assertions for these techniques to be effective remains a major problem. This issue is a significant challenge facing automatic assertion generation [El Mandouh 12, Li 10, Vasudevan 10], which can see an explosion in the number of assertions, many of which are ineffective at catching bugs.

Existing formal verification techniques employing BMC [Lin 15a, Reid 16, Singh 18] have issues in detecting bugs that require a long activation sequence, and the cost of theorem proving as employed in [Kroening 00, Srinivasan 10, Manolios 08] is very high [Bhadra 10]. Other works [Das 02, Gulwani 09, Pandav 05, Su 96, 98, Thalmaier 10] try to learn invariants on the design to be used as constraints, but these techniques tend to be ad-hoc and are not completely automated.

Existing HT detection techniques that can be applied in a pre-silicon verification setting to detect HTs which corrupt design functionality broadly fall into two categories: i) HDL analysis methods; and ii) Formal methods [Xiao 16]. HDL analysis can be done on behavioral (e.g., [Zhang 11]) or structural HDL code (e.g., [Cakır 15, Hicks 10]). The idea here is that signals associated with Trojans are mostly "unused" or "rare" in comparison to signals which are part of the actual design. Different methods use different metrics to separate the "rare" signals from the set of all signals. Some techniques [Cakır 15, Hicks 10, Zhang 11, Zhang 13b] use simulation data along with "rareness" metrics (e.g., code coverage, signal correlation). These techniques require the simulation data to be "good" (i.e., there should be enough data so that only a few signals are marked as "rare" while simultaneously the data should not be "too much" so that HTs are missed), which is difficult to ensure. Techniques like [Waksman 13, Yao 15] do not need simulation data, but also face the issue of trading of false-positives (i.e., spurious detection of HTs) for false-negatives (i.e., failure to detect HTs) and vice-versa, depending on the thresholds set for their "rareness" metrics. Additionally, stealthy HTs have been designed [Zhang 14] to bypass HDL analysis techniques [Hicks 10, Waksman 13, Zhang 11, Zhang 13b]. In contrast, our technique: i) does not need any simulation data; ii) detects HTs given in [Salmani 13, Zhang 13a, Zhang 14]; and iii) does not face the problem of false positives.

Formal methods for finding HTs generally either use BMC [Rajendran 15, 16] or SAT-based equivalence checking [Banga 10, Reece 16, Shrestha 12] or theorem proving [Guo 17, Jin 13, Love 12]. All these techniques need manually crafted properties (and proofs when theorem proving is used) and face the same limitations as traditional formal methods used in pre-silicon verification as discussed before. Specifically, [Banga 10, Rajendran 15, Reece 16, Shrestha 12] fail to find HTs that require a long activation sequence (e.g., an HT that is triggered only when a 128-bit counter reaches its largest value), while our technique finds them. Additionally, SAT-based equivalence checking techniques like [Banga 10, Reece 16, Shrestha 12] need a "golden" model of the design to compare with, which we do not need. Complementary HT works to this work include techniques focusing on detection of HTs which leak sensitive data [Fern 17, Hu 16, Jin 12, Rajendran 16] and HT prevention techniques [Chakraborty 09, Dupuis 14, Samimi 16, Waksman 11].

5. Conclusion

In this paper, we extended Symbolic QED with symbolic starting states to overcome limitations of existing pre-silicon verification techniques for both logic bug and HT detection. We automatically, effectively, and quickly found "difficult" logic bugs and HTs, as demonstrated on open-source RISC-V processor cores. Future research directions include: i) detecting logic bugs and HTs in other components found in an SoC e.g., uncore components, accelerators, and analog/mixed-signal blocks; ii) handling other QED transformations such as CFTSS-V and CFCSS-V [Singh 18]; iii) automatically deriving information needed to write constraints on the symbolic starting state; iv) exploring abstraction techniques to reduce runtimes and further increase the BMC bounds; and v) a complete understanding of the tradeoffs between the symbolic starting states technique developed in this paper and S^2QED [Fadiheh 18].

References

[Extended 18] Extended version of paper, (url TBD).

[ARM] "Cortex-A15," https://developer.arm.com/products/processors/cortex-a/cortex-a15.

[Banga 10] Banga, M., and M.S. Hsiao, "Trusted RTL: Trojan detection methodology in pre-silicon designs," *IEEE Intl. Symp. on HOST*, pp. 56-59, 2010.

[Bergeron 03] Bergeron, J., "Writing Testbenches: Functional Verification of HDL Models," *Kluwer*, 2003.

[Bhadra 10] Bhadra, *et al.*, "A survey of hybrid techniques for functional verification," *IEEE Design & Test of Computers*, vol. 24, no. 2, pp. 112-122, 2007.

[Cakır 15] Cakır, B., and S. Malik, "Hardware Trojan detection for gate-level ICs using signal correlation based clustering," *Proc. DATE*, pp. 471-476, 2015.

[Campbell 15] Campbell, K., *et al.*, "Hybrid Quick Error Detection (H-QED): Accelerator Validation and Debug Using High-Level Synthesis Principles," *Proc. DAC*, pp. 1-6, 2015.

[Chakraborty 09] Chakraborty, R.S., and S. Bhunia, "HARPOON: an obfuscation-based SoC design methodology for hardware protection," *IEEE Trans. CAD*, vol. 28, pp. 1493-1502, 2009.

[Clarke 01] Clarke, E., *et al.*, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design*, Vol. 19, No. 1, pp. 7-34, 2001.

[Dupuis 14] Dupuis, S., *et al.,* "A novel hardware logic encryption technique for thwarting illegal overproduction and Hardware Trojans," *IEEE Intl. On-Line Testing Symp.*, pp. 49-54, 2014.

[El Mandouh 12] El Mandouh, E., and A.G. Wassal, "Automatic Generation of Hardware Design Properties from Simulation Traces," *Proc. ISCAS*, pp. 2317-2320, 2012.

[Fadiheh 18] Fadiheh, M. R., *et al.*, "Symbolic quick error detection using symbolic initial state for pre-silicon verification," *Proc. DATE*, pp. 55-60, 2018.

[Fern 17] Fern, N., I. San, and K.T.T. Cheng, "Detecting hardware Trojans in unspecified functionality through solving satisfiability problems," *Proc. ASP-DAC,* pp. 598-504, 2017.

[Fine 03] Fine, S., and A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," *Proc. DAC*, pp. 286-291, 2003.

[Foster 03] Foster, H., A. Krolnik, and D. Lacey, "Assertion-based Design," *Kluwer Academic Publishers*, 2003.

[Foster 15] Foster, H.D., "Trends in functional verification: A 2014 industry study," *Proc. DAC*, pp. 1–6, 2015.

[Guo 17] Guo, X., *et al.,* "Eliminating the Hardware-Software Boundary: A Proof-Carrying Approach for Trust Evaluation on Computer Systems," *IEEE Trans. on Information Forensics and Security*, vol. 12, no. 2, pp. 405-417, 2017.

[Hicks 10] Hicks, M., *et al.,* "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," *Proc. IEEE Symp. on Security and Privacy*, pp. 159–172, 2010.

[Hong 10] Hong, T., *et al.*, "QED: Quick Error Detection Tests for Effective Post-Silicon Validation," *Proc. ITC,* pp. 1-10, 2010.

[Hu 16] Hu, W., *et al*., "Detecting Hardware Trojans with Gate-Level Information-Flow Tracking," *Computer*, vol. 49, no. 8, pp. 44-52, 2016.

[Jin 12] Jin, Y., and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," *Proc. IEEE VLSI Test Symp.*, pp. 252–257, 2012.

[Jin 13] Jin, Y., and Y. Makris, "A proof-carrying based framework for trusted microprocessor IP," *Proc. ICCAD*, pp. 824–829, 2013.

[Karri 10] Karri, R., *et al.*, "Trustworthy hardware: Identifying and classifying hardware Trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[Kim 04] Kim, Y.I., "Communication-Efficient Hardware Acceleration for Fast Functional Simulation," *Proc. 41st DAC*, pp. 293-298, 2004.

[King 08] King, S.T., *et al.*, "Designing and implementing malicious hardware," *Proc. LEET*, pp. 1–8, 2008.

[Kitchen 07] Kitchen, N., and A. Kuehlmann, "Stimulus generation for constrained random simulation," *Proc. ICCAD*, pp. 258-265, 2007.

[Li 10] Li, W., F. Alessandro, and S.A. Seshia, "Scalable Specification Mining for Verification and Diagnosis," *Proc. DAC*, 2010.

[Lin 12] Lin, D., *et al.*, "Quick Detection of Difficult Bugs for Effective Post- Silicon Validation," *Proc. DAC*, pp. 561-566, 2012.

[Lin 14] Lin, D., *et al.*, "Effective Post-Silicon Validation of System-on-Chips Using Quick Error Detection," *IEEE Trans. CAD*, Vol. 33, No. 10, pp. 1573-1590, 2014.

[Lin 15a] Lin, D., *et al.*, "Quick error detection tests with fast runtimes for effective post-silicon validation and debug," *Proc. DATE*, pp. 1168-1173, 2015.

[Lin 15b] Lin, D., *et al*., "A Structured Approach to Post-Silicon Validation and Debug Using Symbolic Quick Error Detection," *Proc. ITC*, pp. 1-10, 2015.

[Liu 14] Liu, C., *et al.*, "Shielding Heterogeneous MPSoCs From Untrustworthy 3PIPs Through Security-Driven Task Scheduling," *IEEE Trans. on Emerging Topics in Computing*, vol. 2, no. 4, pp. 461-472, 2014.

[Love 12] Love, E., Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Trans. Inf. Forensics Security*, vol. 7, no. 1, pp. 25-40, 2012.

[MIPS 96] Yeager, K, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, 16(2):28–41, 1996.

[Mitra 10] Mitra, S., S.A. Seshia, and N. Nicolici, "Post-Silicon Validation Opportunities, Challenges and Recent Advances," *Proc. DAC*, pp. 12-17, 2010.

[Rajendran 15] Rajendran, J., V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," *Proc. DAC*, Article 112, 6 pages, 2015.

[Rajendran 16] Rajendran, J., *et al*., "Formal Security Verification of Third Party Intellectual Property Cores for Information Leakage," Intl. Conf. on VLSI Design, pp. 547-552, 2016.

[Reece 16] Reece, T., and W.H. Robinson, "Detection of Hardware Trojans in Third-Party Intellectual Property Using Untrusted Modules," *IEEE Trans. CAD*, vol. 35, no. 3, pp. 357-366, 2016.

[Ridecore1] "RIDECORE: RISC-V Dynamic Execution CORE," https://github.com/ridecore/ridecore.

[Ridecore2] "Issue: bugs in rs_mul," https://github.com/ridecore/ridecore/issues/4.

[RISCV] "RISC-V: The Free and Open RISC Instruction Set Architecture," https://riscv.org/.

[RISCVP] "RISC-V Privileged Architecture," https://people.eecs.berkeley.edu/~krste/papers/riscv-privileged-v1.9.pdf.

[Salmani 13] Salmani, H., M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," *Proc. ICCAD*, pp. 471–474, 2013.

[Samimi 16] Samimi, M.S., *et al.,* "Hardware enlightening: No where to hide your Hardware Trojans!" *IEEE Intl. Symp. on On-Line Testing and Robust System Design*, pp. 251-256, 2016.

[Shrestha 12] Shrestha G., and M.S. Hsiao, "Ensuring trust of third-party hardware design with constrained sequential equivalence checking," *IEEE Conf. on Tech. for Homeland Security*, pp. 7-12, 2012.

[Shyam 06] Shyam S., and V. Bertacco, "Distance-Guided Hybrid Verification with GUIDO," *Proc. DATE*, pp. 1211-1216, 2006.

[Singh 18] Singh, E., *et al.*, "Logic Bug Detection and Localization Using Symbolic Quick Error Detection," *IEEE Trans. CAD*, 2018.

Appendices

A. Bug and HT scenarios

In the following tables, we give the different logic bug and HT scenarios that were tested in Table 1 of Sec. 3.

Table A.1.a. Activation criteria for "long" logic bugs.

Processor Core	1. Data forwarding between pipeline stages.
	2. Two specific instructions within *X* cycles.
	3. *R* registers must each contain a specific value *V*.
	4. A specific sequence of *N* instructions must execute within *Y* cycles.
	5. A specific cache state.

Table A.1.b Bug effect from [Singh 18].

Processor Core	1. Next instruction corrupted to NOP.
	2. Next instruction opcode incorrectly decoded.
	3. Next instruction register read corrupted.

Table A.2.a Activation criteria for Hardware Trojans from [Salmani 13].

Processor Core	1. Specific sequence of length N appears on M_1 internal wires.
	2. X_1 bit counter reaching a specific value.
	3. Comparator on M_2 internal wires becomes true.
	4. X_2 bit rare event counter reaching a specific value.

Table A.2.b Trojan effects

Processor Core	1. An in-flight instruction changed to NOP.
	2. Opcode of an in-flight instruction changed.
	3. Next register read corrupted.
	4. Next result of an execution unit changed.
	5. Corrupts ROB to prematurely commit next instruction.

Table A.2.c. Trojan Design Techniques

Methodology	Trojan stealthy against following detection techniques
[Salmani 13]	Traditional pre-silicon verification techniques
[Zhang 13a]	UCI [Hicks 10] and techniques based on coverage metrics [Hicks 10, Zhang 11]
[Zhang 14]	[Hicks 10, Zhang 11], FANCI [Waksman 13], VeriTrust [Zhang 13b]

B. Background

B.1. QED and the EDDI-V Transformation

QED is a testing technique which takes existing system validation tests and automatically transforms them into a set of new tests using various QED transformations [Lin 12, Lin 14]. QED tests have been demonstrated to be highly effective for quickly detecting logic and electrical bugs inside processor cores, uncore components, accelerators, and components related to power-management features [Campbell 15, Hong 10, Lin 12, 14, 15a]. Here we the EDDI-V transformation, which is the focus of this work.

EDDI-V [Hong 10, Lin 14] targets bugs inside processor cores by frequently checking the results of *original* instructions against the results of *duplicate* instructions. First, the registers and memory space are divided into two halves[7], one for the original instructions and one for the duplicated instructions. Next, corresponding registers and memory locations for the original and the duplicated instructions are initialized to the same values. Then, for every load, store, arithmetic, logical, shift, or move instruction in the original test, EDDI-V creates a corresponding duplicate instruction that performs the same operation, but uses the registers and memory reserved for duplicate instructions. The duplicated instructions execute in the same order as the original instructions.

The EDDI-V transformation also inserts periodic check instructions that compare the results of the original instructions against those of the duplicated instructions. Each check instruction is of the form:

$$\mathrm{CMP\ Ra, Ra'},$$

where Ra and Ra′ are the original and (corresponding) duplicate registers, respectively. A mismatch in any check instruction indicates an error.

B.2. Symbolic QED

Symbolic QED [Lin 15a, Singh 18] combines QED transformations with *Bounded Model Checking* (BMC). Bounded Model Checking uses a Boolean Satisfiability (SAT) solver to check whether a given property can be violated within a bounded number of cycles for a given RTL design, starting from a concrete state. If a solution is found, a counter-example (a concrete trace violating the property, i.e., a bug trace) is produced. BMC guarantees that if a counter-example is found, it is a minimal-length counter-example [Clarke 01]. Symbolic QED creates a BMC problem that searches through *all possible* EDDI-V tests. The BMC tool searches for counter-examples to properties of the form:

$$\mathrm{Ra} == \mathrm{Ra}',$$

where Ra is an original register and Ra' is the corresponding duplicate register written to during an EDDI-V test. Without additional constraints, the BMC engine will find spurious counter-examples. For example, the instruction sequence MOV R1, 1; MOV R17, 2; CMP R1, R17 results in $\mathrm{R1} \neq \mathrm{R17}$. The inequality is not caused by a real bug. To avoid such situations, we require all counter-examples to be *QED-compatible*. We define a *QED-compatible* bug trace as a sequence of inputs with the following properties: 1. Inputs must be valid instructions. Specifications of valid instructions can be obtained from the Instruction Set Architecture (ISA) of the processor core; 2. The instruction sequence is an EDDI-V test; 3. The original and duplicate instruction subsequences must execute in the same order. 4. The comparison between an original register R and its corresponding duplicate register R' occurs only if the original and corresponding duplicate instructions have both committed.

Ensuring that only QED-compatible bug traces are considered by BMC requires constraining the inputs to the design. This is done by adding a *QED module* to the fetch stage of the processor core during BMC. The QED module automatically transforms a sequence of original instructions into a QED-compatible sequence e.g., as in Fig. 5. The QED module only requires that the input sequence is made up of valid instructions that read from or write to only the registers and memory allocated for the original instructions (conditions that can be specified directly to the BMC tool). After execution, a signal is asserted (*qed_ready*) when the original and corresponding duplicated registers should contain the same values under bug-free situations, i.e., the BMC tool should check the property:

$$\mathit{qed\_ready} \rightarrow \bigwedge_{a\in\{0..\frac{N}{2}-1\}} \mathrm{Ra} == \mathrm{Ra}',$$

where N is the number of registers defined by the ISA. Here (for $a \in \{0..N/2-1\}$), Ra and Ra' correspond to registers allocated for original instructions and duplicated instructions, respectively.

```
R1 = R1 + 5
R2 = R2 – R1
R3 = R1 * R2

(a)
```

```
R1 = R1 + 5
R2 = R2 – R1
R3 = R1 * R2

R16 = R16 + 5
R18 = R18 – R16
R19 = R16 * R18

(b)
```

Fig. 5: Example of QED transformation by the QED module. (a) A sequence of original instructions, and (b) transformed instructions executed by the processor.

The starting state for the BMC run must also be a *QED-consistent* state, in which the value of each original register or memory location matches the corresponding duplicate register or memory location. This is to prevent false counter-examples from being generated. One way to obtain such a state is to run an EDDI-V test in simulation and stop immediately after QED checks have compared all register and memory values.

B.3. S^{2}QED

S^{2}QED [Fadiheh 18] is a technique which extends Symbolic QED by incorporating symbolic initialization. Like the technique in this paper, S^{2}QED also focuses strictly on EDDI-V tests. S^{2}QED instantiates two copies of the CPU (call them CPU 1 and CPU 2). An arbitrary one-to-one mapping is then determined between the registers of the two CPU instances. A new notion of "QED consistency" is defined when all values in the registers of CPU 1 match the values in the corresponding mapped registers of CPU 2. Then, CPU 1 is constrained to start from a concrete flushed-pipeline state, and CPU 2 starts from a symbolic initial state, with the additional constraint that the two CPUs satisfy this notion of QED consistency in the initialization.

Then, at the start of verification, CPU 1 fetches a valid instruction (called the instruction under verification, or IUV) while CPU 2 fetches the corresponding QED duplicate instruction (i.e., same instruction with each register of CPU 1 replaced by its corresponding mapped register in CPU 2). Then, CPU 1 is constrained to fetch NOPs until the IUV commits, while CPU 2 can fetch arbitrary valid instructions (treated like symbols by the formal tool). S^{2}QED then attempts to prove that at commit time for the IUV, the registers in the two CPUs will always remain QED consistent.

The technique developed in this paper differs from S^{2}QED in a few key aspects: 1) S^{2}QED does not require a QED module or QED recorders during pre-silicon verification, whereas the technique developed in this paper does; 2) S^{2}QED requires duplication of the CPU (only during pre-silicon verification), whereas the technique developed in this work requires only a single CPU; However, to motivate the technique presented in this work, the following is an example of a bug that can only be caught by the technique developed in this paper, and not S^{2}QED.

Example. *When all the registers in the register file contain the value -1, the next register write is corrupted.*

Because this bug requires that all registers share the same value, all possible register mappings between CPU 1 and CPU 2 maintain QED-consistency during the bug activation. For any IUV, the bug effects both CPUs identically, and no difference can be detected between the final values in the corresponding mapped registers. In contrast, the technique developed in this paper makes original and duplicate instructions execute in sequence. This can allow activation of the bug during either an original or duplicate instruction, but not the other, so the bug can be detected.

B.4. New QED module for Single Processor Cores

Here, we describe the QED module we used for running Symbolic QED on single processor cores. This new QED module differs from the QED module used in [Singh 18], and we show (see Sec. 3) that there are specific bugs that can only be caught by this new QED module but not by the previous one. Pseudocode for the QED module is given in Fig. 6(a). Inputs are: 1) *enable*, which disables the QED module if 0; 2) *next_instruction*, which is the next chosen sequential instruction to be executed; 3) *fetch_next*, which is high when the core is ready to receive an instruction (i.e., fetch stage is not stalled); 4) *original*, which tells the module to execute an original instruction (when high), or a duplicate instruction (when low). The outputs from QED module are: 1) *instruction_valid*, which indicates whether the output instruction is valid or not; and 2) *instruction_out*, which gives the instruction to be executed.

The QED module has internal variables: 1) *queue*, which is a queue data structure used to store previously seen original instructions that have not yet been executed in the duplicate subsequence; 2) *head_instruction*, which stores the previous head of the queue; 3) *insert_valid*, which is true when the QED module can execute an original instruction; 4) *delete_valid*, which is true when the QED module can execute a duplicate instruction; 5) *duplicate_instruction,* which gives the next instruction in the duplicate subsequence to be executed (only used when *original* is 0).

To run Symbolic QED, we also need to determine when it is safe to assert QED checks, i.e., the logic for the *qed_ready* signal. Pseudocode for determining *qed_ready* for the QED module in Fig. 6(a) is given in Fig. 6(b). To avoid having false fails, we only assert QED checks when an equal number of commits (writes) have been made to registers mapped to original instructions and to registers mapped to duplicate instructions. This can be accomplished by keeping track of the number of original and duplicate instruction commits to the register set as shown in Fig. 6(b). For simplicity, in Fig. 6(b), we assume that at most one instruction can commit per cycle. For superscalar processors that can commit multiple instructions in the same cycle, we simply keep track of all corresponding pairs of *write_valid* (which tells whether the input data to be written is valid) and *write_address* (which signifies the address for the data to be written) signals, have a separate *is_original* signal (which identifies if an address corresponds to an "original" or "duplicate" location) for each address, and allow the original and duplicate counters to each be incremented multiple times based on the evaluation of each of the *is_original* signals

In contrast, the QED module of [Singh 18] requires that all original instructions complete, a waiting period occurs for the pipeline of the core to be flushed, and finally the duplicate instructions are executed, before the *qed_ready* signal is asserted. This QED module instead allows arbitrary interleaving (by simply giving control of the *original* input of Fig. 6(a) to the BMC tool) of the original and duplicate instruction subsequences without requiring the waiting period before QED checks.

The QED ready enable logic in Fig. 6(b) can be further enhanced:

1. The current QED ready enable logic is only applicable to single processor cores, since a multi-core system would require modification of the *qed_ready* logic to consider the original and duplicate commits across all cores. This can be challenging in situations where multiple cores operate with a shared address space, since QED checks will require cache coherency to avoid false positives. For simplicity, we do not consider this situation in this paper.

2. For some processors, e.g., superscalar processors with explicit register renaming (MIPS 10000 [MIPS 96] and ARM's Cortex-A15 [ARM]) the designation of original or duplicate instruction cannot be made solely on where they write (unlike in Fig. 6(b)). This issue can be corrected easily, by including the current state of the register mapping table as an input to the function is_write_to_original_space. Further, each time a QED check happens, the same mapping table must be used to map each logical address to its current physical address before comparing "original" and "duplicate" values. The RISC-V cores used in this work, however, do not have this issue.

```
INPUT: enable, next_instruction, fetch_next, original
OUTPUT: instruction_out, instruction_valid
// initialization
queue ← 0;  head_instruction ← 0;
// end initialization
insert_valid ← fetch_next & original & ~queue.is_full();
delete_valid ← fetch_next & ~original & ~queue.is_empty();
instruction_valid ← insert_valid | delete_valid;
if insert_valid then
  queue.push(next_instruction);   // store this instruction in queue
else if delete_valid then
  head_instruction ← queue.pop();  // remove instruction at the head from queue
end if
duplicate_instruction ← create_duplicated_version(head_instruction);
instruction_out ← (enable & ~original) ? duplicate_instruction : next_instruction;
```

(a)

```
INPUT: write_valid, write_address
OUTPUT: qed_ready
// initialization
qed_ready ← false;  count_original ← 0;  count_duplicate ← 0;
// end initialization
is_original ← is_write_to_original_space(write_address);
if write_valid then
  if is_original then
    count_original++;       // increment number of original instructions committed
  else
    count_duplicate++;     // increment number of duplicate instructions committed
  end if
end if
qed_ready ← (count_original == count_duplicate) ? true : false;
```

(b)

Fig. 6. Pseudo code for new single-core QED module. (a) QED module, and (b) QED ready enable logic

C. Assumptions on Bug-Free Designs

In this Appendix, we detail our assumptions on how any bug-free design should operate. We relate these assumptions to the QED constraints of Sec. 3., and explain how they are used to prevent false positives.

Assumption-1: Let I_1 and I_2 be two Symbolic QED instructions given abstractly as:

$$I_1 : d \leftarrow \text{op } s_1, s_2, \dots, s_m$$
$$I_2 : d' \leftarrow \text{op } s'_1, s'_2, \dots, s'_m.$$

Here op represents an operation performed on data stored in the operand locations (s_1, s'_1 etc.), and the instructions write the computed result to a destination location in the architectural state (e.g., d, d'). Let $data(s_1, I_1)$ be a notation to represent the operand s_1 data used by instruction I_1 for computing the result, (Note the actual data for the source operand may be either obtained from an architectural state, e.g., architectural register, or a micro-architectural state (e.g., by data forwarding). $data(s, I)$ abstracts away these extra details and only represents the data value used by instruction I for operand s.) and $val(T_{I_1}, d)$ represents the value in the architectural location d immediately after instruction I_1 commits at time T_{I_1} (i.e., the value written by instruction I_1 to location d).

Now, Assumption-1 states: If two Symbolic QED instructions I_1, I_2 have the same op and $\forall i \in \{1, \dots, m\}, data(s_i, I_1) = data(s'_i, I_2)$, then $val(T_{I_1}, d) = val(T_{I_2}, d')$.

This Assumption simply states that when a Symbolic QED instruction executes twice on the same data, then it should result in the same outputs.

Assumption-2: Let I_1 and I_2 be two Symbolic QED instructions given as:

$$I_2 : s_i \leftarrow op_2 \; s'_1, s'_2, \dots, s'_m$$
$$I_1 : d \leftarrow op_1 \; s_1, s_2, \dots, s_m,$$

where $i \in \{1, \dots, m\}$ and I_2 is the last Symbolic QED instruction with which I_1 has Read-after-Write (RAW) dependency for operand s_i. Then: if I_1, I_2 are as above, $data(s_i, I_1) = val(T_{I_2}, s_i)$.

Further, if there are no earlier Symbolic QED instructions writing to an operand s_i of I_1, where $i \in \{1, \dots, m\}$, then $data(s_i, I_1) = val(T_C, s_i)$.

In words, this Assumption states that when any operand of a Symbolic QED instruction has RAW dependency with any earlier Symbolic QED instruction(s), it should obtain the correct source value (which is the result of the last instruction it is dependent on). Also, when an operand of a Symbolic QED instruction does not have dependencies with earlier Symbolic QED instructions, it obtains the correct source value from the architectural state at time T_C, to be used in its computation.

Both Assumptions are naturally satisfied in a bug-free processor core, given it starts from a state which is reachable from one of the specified reset states. However, these Assumptions may not hold if the core starts from an arbitrary symbolic state. Nevertheless, as Constraint C-1 guarantees that all SIF instructions commit at T_C, we expect them to hold after T_C. In other words, even if the processor pipeline starts from an unreachable state, we expect the pipeline to reach a state (after T_C) at which the Assumptions hold. For example, if the processor core has an in-order pipeline, then we expect the state-bits associated with the in-order pipeline stages to go to a reachable state after T_C, as all the SIF instructions would have been completed by then and the pipeline stages are either empty or filled with Symbolic QED instructions (which are valid instructions that have propagated through the pipeline stages).

Thus, we expect the Assumptions to hold for in-order pipeline cores. For superscalar cores also, Constraint C-1 guarantees that no SIF instructions are in the processor pipeline, which is typically made of in-order stages for fetch, decode, and commit and an out-of-order execution stage having reservation stations or instruction buffers to store in-flight instructions. At T_C, the state bits associated with in-order pipeline stages, along with state bits of some entries in the instruction buffers will be filled with Symbolic QED instructions, taking these state bits to valid (reachable) states. Also, at T_C, all remaining entries in the instruction buffers should be empty entries, otherwise, it would mean that there is an uncommitted SIF instruction which would violate Constraint C-1. Thus, the state bits associated with these entries, at T_C, must be of no consequence to the execution of the processor. Thus, the overall state of a superscalar core, at T_C, is such that it has only valid Symbolic QED instructions in-flight. Consequently, we expect the Assumptions to also hold in a typical superscalar core. We have empirically seen that the Assumptions hold in both an in-order core [vscale] and a superscalar core [Ridecore1].

D. Specifying QED Constraints to the BMC Tool

In this Appendix, we describe in detail how we specify the QED constraints on the symbolic starting state (introduced in Sec. 2.1.) to the BMC tool.

D.1. Specifying C-1

Constraint C-1 states that by T_C, all SIF instructions have committed, while no Symbolic QED instruction has committed. This condition is naturally satisfied in processors with in-order execution, which is the case for processors with in-order pipelines. But, this is not the case with superscalar cores. This is due to instruction indirection, i.e., renaming of instructions using respective ROB entries, which happens to support out-of-order execution inside a superscalar core. As we are starting with a symbolic state, ROB entry locations for SIF instructions may be chosen by the BMC tool such that they commit after Symbolic QED instructions commit[5], thereby violating C-1. So, in a superscalar core, specifying C-1 to the BMC tool involves properly constraining the ROB entries for SIF instructions to avoid the issue.

D.2. Specifying C-2

Constraint C-2 states that: at T_C, the processor state is QED consistent. Further, after T_C, only Symbolic QED instructions can write to architectural state (test modes such as scan need to be turned off). We can specify C-2 to the BMC tool using the below statements:

$$\uparrow(\mathrm{SIF}_{\mathrm{complete}}) \rightarrow \forall (\mathrm{i},\mathrm{j}) \in \mathrm{M}_{\mathrm{r}}, (\mathrm{R}_{\mathrm{i}} = \mathrm{R}_{\mathrm{j}})$$

$$\uparrow(\mathrm{SIF}_{\mathrm{complete}}) \rightarrow \forall (\mathrm{i},\mathrm{j}) \in \mathrm{M}_{\mathrm{m}}, (\mathrm{Mem}_{\mathrm{i}} = \mathrm{Mem}_{\mathrm{j}})$$

$$.\uparrow(\mathrm{Clock}) \rightarrow \text{ if } (\mathrm{SIF}_{\mathrm{complete}})\ \mathrm{Test}_{\mathrm{enable}} = 0$$

Above, $\uparrow$(signal_name) is true when signal_name transitions to high from low, and M_r (M_m) is the set of all mapped (original, duplicate) pairs of registers (memory locations). Importantly, if there are multiple test modes, the $\mathrm{Test}_{\mathrm{enable}}$ signal for all of them need to be set to low. These statements are specified to the BMC tool by writing them in the form of *assume statements* (e.g., in System Verilog). Note that these constraints take the same form for both in-order and superscalar cores.

D.3. Specifying C-3

Constraint C-3 states that: All Symbolic QED operands, must have one of the following properties: i) if the operand has already read source data then this data must match the corresponding register/memory location data at T_C; or ii) if the operand is waiting at T_C, then it is waiting on an earlier Symbolic QED instruction.

Constraint C-3(ii) is vacuously true for in-order pipelines, as an instruction only makes progress when all its operands have already read their respective data, otherwise the instruction just stalls for operand data. We use the information obtained from the Symbolic QED operand recorder (Fig. 4) to specify C-3 to the BMC tool using the below statements:

$$\uparrow(\mathrm{SIF}_{\mathrm{complete}}) \rightarrow \forall \mathrm{s} \in \mathrm{src1\_buffer}\ (\mathrm{s.data} == \mathrm{Reg[s.addr]})$$

$$\uparrow(\mathrm{SIF}_{\mathrm{complete}}) \rightarrow \forall \mathrm{s} \in \mathrm{src2\_buffer}\ (\mathrm{s.data} == \mathrm{Reg[s.addr]})$$

$$\uparrow(\mathrm{SIF}_{\mathrm{complete}}) \rightarrow \forall \mathrm{m} \in \mathrm{mem\_buffer}\ (\mathrm{m.data} == \mathrm{Mem[m.addr]}).$$

Above, *s.data* gives data stored for entry *s* in the buffer, while *s.addr* gives the address. *Reg* represents the architectural register array while *Mem* represents the architectural memory array. These statements are specified to the BMC tool by writing them in the form of *assume* statements (e.g., in System Verilog). For a superscalar core, specifying Constraint C-3(i) is the same as above, but we need additional information, as discussed in Sec. 3.2.2. to specify Constraint C-3(ii) to the BMC tool.

D.4. Finding Counter-Examples using BMC

The QED property (see Appendix B.2.) used by BMC to find counter-examples in Symbolic QED is modified to support symbolic starting states and is given as below:

$$qed\_ready\ \&\ SIF\_complete \rightarrow \bigwedge_{a\in\{0,\ldots,\frac{N}{2}-1\}} \mathrm{Ra} == \mathrm{Ra}',$$

here the only change is the addition of the $SIF\_complete$ (as given in Sec. 2.2.1.) precondition to the QED property.

[7]For EDDI-V, if it is not possible to divide the registers into two halves (i.e., if the original test needs to use all the available registers), we can use memory to store the register values. The details are in [Lin 14].