Symbolic Quick Error Detection Using Symbolic Initial State for Pre-Silicon Verification

Mohammad Rahmani Fadiheh*, Joakim Urdahl*, Srinivas Shashank Nuthakki†,
Subhasish Mitra†, Clark Barrett†, Dominik Stoffel*, Wolfgang Kunz*

*Dept. of Electrical and Computer Engineering
Technische Universität Kaiserslautern, Germany

†Depts. of Electrical Engineering and Computer Science
Stanford University, Stanford, CA, USA

***Abstract*—Driven by the demand for highly customizable processor cores for IoT and related applications, there is a renewed interest in effective but low-cost techniques for verifying systems-on-chip (SoCs). This paper revisits the problem of processor verification and presents a radically different approach when compared to the state of the art. The proposed approach is highly automated and leverages recent progress in the field of post-silicon validation by the method of Quick Error Detection (QED) and Symbolic Quick Error Detection (SQED).**

In this paper, we modify SQED by incorporating a symbolic initial state in its BMC-based analysis and generalize the approach into the S^{2}QED method. As a first advantage, S^{2}QED can separate logic bugs from electrical bugs in QED-based post-silicon validation. Secondly, it also makes a strong contribution to pre-silicon verification by proving that the execution of each instruction is independent of its context in the program. The manual efforts for the proposed approach are orders of magnitude smaller than for conventional property checking. Our experimental results demonstrate the potential of S^{2}QED using the Aquarius open-source processor example.

***Index Terms*—formal verification, post-silicon validation, Quick Error Detection, S^{2}QED.**

I. Introduction

While today's techniques for *pre-silicon verification* account for more than 50% of the total efforts in the System-on-Chip (SoC) design cycle [1], they often are still inadequate for detecting difficult design bugs before tape-out [1], [2], [3]. These bugs have to be found in *post-silicon validation* and must be fixed or bypassed by patching or re-spinning which drastically increases design time and costs [3].

With the increasing complexity of microarchitectures and the growing demand for highly optimized and individually customized processors, for example for Internet of Things (IoT) applications, processor verification is an increasing challenge in SoC design. In this paper, we address the renewed need for low-cost but highly effective processor verification techniques and propose a radically new approach for this purpose.

Processors are specified in programmer-level models, called instruction set architecture (ISA). An ISA describes the instructions as independent operations of the system that execute separately from each other; however, the implemented microarchitecture usually executes several instructions in parallel with the help of static and dynamic pipelining, and related concepts. As a result of the complex interrelations between different instructions executed in a sequence, in modern microarchitectures it is a major challenge to prove that the correct execution of each instruction is independent of other instructions in the pipeline and of the previous program history. In other words, it must be shown that the semantics of an instruction does not depend on its context in the program but is uniquely defined in the given implementation. This proof, however, is not at all a trivial task. The errata sheets published for modern processors attest to this and often describe dozens of functional bugs related to this problem. "Work-arounds" are proposed that must be taken into account by the programmer or must be implemented in the compiler. In such cases, the "work-around" usually consists in restricting the use of an instruction to a specific context in which its behavior is correct. One of the main contributions of this paper is that functional bugs of this category will be avoided by the proposed approach.

To this date, industrial practice in processor verification heavily relies on simulation. Most techniques are based on code coverage, signal toggle coverage or similar notions and fall short of exploring the state space completely. Since many difficult bugs in microprocessors, as described above, are related to control logic and depend on the state of the system [4], they may escape simulation and remain undetected until post-silicon validation.

A tremendous amount of research has been carried out in the field of *formal* processor verification over the years. We sketch only briefly some cornerstones, many of them dating back for at least a decade. Dill and Burch used a quantifier-free logic containing uninterpreted functions to verify a pipelined processor [4], [5]. Their work inspired a lot of other research abstracting from the complexity of the datapath combined with different Boolean methods such as symbolic model checkers [6], [7] and BDD-based decision procedures [8]. These methods usually work on abstract models of a processor. However, most of the difficult bugs appear in the RTL description of the design [9] and can be missed easily in an abstract model.

Numerous approaches have been proposed in the past applying SAT-based formal verification [10] to SoCs and processors. In a practical study by [11] it was shown that SAT-based property checking techniques can "completely" [12], [13] verify an RTL description against its formalized ISA specification. While this can eliminate all functional design bugs from the processor's errata sheets, the manual effort for such approaches is significant and is reported to be around 2,000 lines of code (LoC) per person month [11]. A more automatic SAT-based approach to verify RTL descriptions of microprocessors is proposed by [9]. While it can be carried out with a lower amount of manual work compared to standard property checking, it lacks the coverage obtainable by property checking. Using Bounded Model Checking (BMC) [14] as its proof method it is effective in finding counterexamples, however, it is not adequate to allow for a valid conclusion about the absence of bugs.

Motivated by the renewed interest in low-cost solutions for highly effective processor verification, as described above, this paper takes a fresh look at the problem, leveraging current trends and developments in the field of post-silicon validation. Symbolic Quick Error Detection (SQED) [15], [16] is a new and completely different approach to tackle the verification of SoCs. This method, which is primarily designed for bug localization in post-silicon validation, is also useful for pre-silicon verification. It is based on Quick Error Detection (QED) software tests [17] and employs BMC as its proof method. SQED tries to find the shortest possible QED tests which expose functional design bugs in the design.

Despite its effectiveness for post-silicon bug localization and bug hunting in pre-silicon verification, SQED does not provide a well-defined conclusion about the functional correctness of the design or the absence of certain classes of bugs. This is not only an issue in pre-silicon verification but also causes

978-3-9819263-0-9/DATE18/©2018 EDAA

problems in post-silicon validation. If a functional design bug, also called "logic bug", escapes pre-silicon verification, this makes the bug localization for a failed QED test much harder, since post-silicon validation needs to clearly distinguish between electrical and logic bugs to find the root cause of a failure.

In this paper, we propose S^{2}QED (Symbolic initial state Symbolic Quick Error Detection) for detecting logic bugs in processors. It is based on extensions and modifications of SQED and uses a symbolic initial state in its analysis to increase the generality of its results.

These are the key characteristics of S^{2}QED:

1) As shown in this paper for static processors, S^{2}QED formally proves the absence of all logic bugs that are detectable by a certain class of QED tests [17]. Hence, in post-silicon validation any failed QED test (of this class) is guaranteed to be caused by an electrical bug, provided that the design passed S^{2}QED in pre-silicon verification.

2) S^{2}QED provides a well-defined contribution to pre-silicon processor verification. It proves that every instruction executes in the same way, independently of its context in the program, thus, guaranteeing a uniquely defined instruction semantics.

3) S^{2}QED is automated to a large extent and requires only little "white-box" information about the design.

4) S^{2}QED unlike other processor verification techniques does not rely on executable specification models and does not require a formal specification of the ISA model.

The usefulness of this method is demonstrated in this paper by verifying the open-source Aquarius processor, a 5-stage in-order pipeline processor based on the SH2 ISA. As will be shown, S^{2}QED is capable of proving correctness of the processor with respect to QED-detectable bugs in a reasonable time. It also detects bugs which are not detectable by SQED. The amount of manual work for this method is found to be much lower than for state-of-the-art property checking.

The rest of the paper is organized as follows. Sec. II briefly reviews QED and SQED. A first approach of generalizing SQED by any-state proofs is discussed in Sec. III. Sec. IV provides a detailed explanation of the S^{2}QED method. Experimental results are reported in Sec. V, followed by concluding remarks in Sec. VI.

II. Background

A. Quick Error Detection

Quick Error Detection (QED) relies on a set of software transformations which transform an existing program into QED tests by inserting various check instructions to reduce the error detection latency of bugs. Error Detection using Duplicated Instructions for Validation (EDDI-V) is one of these software transformations which targets bugs inside the processor. In this paper, we focus on EDDI-V tests.

EDDI-V partitions the register file into two sets, called original registers and duplicated registers, with a unique correspondence mapping between the two sets. In the same way, the memory space of the program is duplicated and a mapping between corresponding software variables is set up. At the beginning of the program, each pair of corresponding registers (memory locations) must be initialized with the same data. EDDI-V transforms an existing software program into a QED test by duplicating sequences of instructions such that the original sequence only works on one half of the register file (memory space) and the duplicate sequence works on the other half. At the end of each sequence, any mismatch between the computation results by the original and the duplicated sequence indicate an error and a bug in the design. This is checked by a short sequence of check instructions that compare values and branch to an error handler upon a mismatch.

```
MOV R0, #1       // bug activation - step 1
MOV R0, #2       // bug activation - step 2
MOV R1, #3       // bug occurs: wrong decoding as NOP
MOV R16, #1      // (bug not activated for R16)
MOV R16, #2      // (bug not activated for R16)
MOV R17, #3      // this executes correctly
CMP R0, R16      // R0=R16, consistent register pair
BNE qed_error    // branch not taken
CMP R1, R17      // R1≠R17, inconsistent registers
BNE qed_error    // branch taken, bug detected
```

Fig. 1. Example of an EDDI-V test seqeunce

Fig. 1 shows an example of such a QED test sequence. Registers $R_{16} \ldots R_{31}$ are the duplicate versions of registers $R_0 \ldots R_{15}$. Let us assume that there is a bug in the pipeline that is activated when there are two instructions in the IF and ID stages, respectively, which write two different values to register R0. Assume that, as an effect of the bug, the next instruction is wrongly decoded as a NOP. The shown instruction sequence is able to detect the bug, as described in the comments behind the individual instructions. In a bug-free processor, after the duplicate sequence , all corresponding register pairs (and memory locations) must have the same contents. We speak of a *QED-consistent register state*.

In the following, we restrict our description to the register file. (Modeling of the memory space in EDDI-V tests is analogous.) As a preparation for the following discussion, let us define a register correspondence between a set of original registers, O, and a set of duplicate registers, D. The register mapping is a bijective function $m : O \mapsto D$. For example, on a processor with N registers, we may define $O = \{R_0, R_1, \ldots, R_{N/2-1}\}$, $D = \{R_{N/2}, R_{N/2+1}, \ldots, R_{N-1}\}$ and $m(R_i) = R_{i+N/2}$. A QED-consistent register state is characterized by the named logic expression:

$$\textit{qed\_consistent\_registers} := \bigwedge_{r \in O} (r = m(r)) \qquad (1)$$

Definition 1. (QED-EDDI-V-detectable) Consider an instruction sequence of arbitrary but finite length which executes on the original register set O starting from some initial state that is QED-consistent, as defined in Eq. 1. Consider a duplication of this instruction sequence which executes on the duplicated register set D correspondingly, as defined by m. The duplicated sequence is executed in an arbitrary interleaving with the original sequence. A logic bug is called *QED-EDDI-V-detectable* if and only if there exists a pair of such instruction sequences such that the resulting processor state violates QED consistence, as defined in Eq. 1, after the last register-write in any of the two sequences has been completed. □

In the following, for reasons of simplicity, we will speak of QED and "QED-discoverable" bugs when we actually refer to QED EDDI-V tests and bugs that can be detected by QED EDDI-V tests, respectively.

B. Symbolic Quick Error Detection (SQED)

SQED [15], [16] combines the QED software test with Bounded Model Checking (BMC) to find instruction sequences exposing a bug in the design. The final goal of this approach is to find a *QED-compatible trace* to the error of minimal length using BMC. A QED-compatible trace is a QED test beginning at a QED-consistent register state.

In order to use BMC, we need a model of the system (i.e., RTL code of the hardware), a property to be proven and an initial state from which the unrolled model starts. For SQED, the solver needs to be controlled such that it only considers QED-transformed instruction sequences. This is achieved by instrumenting the hardware description of the processor with a special module called *QED module*, which is not added to the manufactured IC but only used for verification. It is inserted between the fetch and the decode unit and it works

TABLE I
EXAMPLE OF A BUG WITH A LONG ACTIVATION SEQUENCE

Bug activation scenario	14 registers in the general purpose register file have a particular value, e.g., -1, and there are two consecutive writes to the same register in two consecutive clock cycles.
Bug efffect	The second write is dropped.

```
assume:
    at $t$: $qed\_ready$;
prove:
    at $t$: $qed\_consistent\_registers()$;
```

Fig. 2. SQED property (Eq. 2) formulated as interval property

as follows: It buffers instructions (and, at the same time, feeds them to the decode unit) until the first branch instruction is encountered. Here it switches to "duplication mode". It replays the instructions from its buffer, thereby translating all register accesses (and memory accesses) to the duplicated registers (and memory locations). At the end of the duplicated sequence, the QED module asserts a flag (*qed_ready*) indicating that the results of both sequences are observable and ready for comparison in the register set. With the help of the QED module, only QED-compatible bug traces are considered. The property to be checked by the BMC solver is:

$$qed\_ready \rightarrow qed\_consistent\_registers \tag{2}$$

where *qed_ready* is the flag asserted by the QED module at the end of the sequence.

BMC starts from a specific state and tries to reach a state violating the property within a finite time window (k_{BMC} clock cycles). In SQED, the initial state for BMC needs to be a QED-consistent register state as mentioned above. One way of obtaining a reachable QED-consistent state is by simulating a QED test and extracting the register and memory values immediately after the check instructions.

Although SQED is capable of detecting some difficult bugs which cannot be detected by conventional methods [15], [16], SQED *cannot prove the absence* of any class of bugs such as EDDI-V-detectable logic bugs, and there are certain bugs that can escape. This is due to the fact that the solver tries to find a bug trace starting from a specific starting state within a finite time window (k_{BMC} clock cycles). For practical designs, it is not feasible to consider a time window that is large enough to cover the sequential depth of the design.

Tab. I describes an example of a bug which is not detectable by SQED. A long sequence of instructions (and a large time window for the solver) is required to load the corresponding values into the register file and to activate the bug; however, it is usually not feasible for the solver in SQED to explore all possible programs involving such long instruction sequences.

It is worthwhile to explore how the problem can be overcome by using a variant of BMC called Interval Property Checking (IPC) [12]. Like in conventional BMC, IPC unrolls the sequential circuitry for a finite number of clock cycles (as given by the property) and checks the validity of the property using SAT. In contrast to standard BMC, however, the starting state of the unrolling is left as free input ("any-state proof").

In some cases, however, spurious counterexamples can occur and the proofs need to be strengthened by invariants [12]. Commercial verification tools usually provide proof engines to generate such invariants automatically.

III. IPC-BASED EXTENSIONS TO SQED

Fig. 2 shows the SQED property of Eq. 2 written as an interval property in pseudo-code. The assumption of the property refers to the output signal *qed_ready* of the QED module. The macro *qed_consistent_registers*() in the proof commitment instantiates the logic expression of Eq. 1.

```
assume:
    at $t$:            $qed\_ready$;
    at $t$:            $flushed\_pipeline()$;
    at $t$:            $qed\_consistent\_registers()$;
prove:
    during [$t$+1, $t$+$k$]:    if $qed\_ready$
                             then $qed\_consistent\_registers()$;
```

Fig. 3. Preliminary SQED property avoiding spurious counterexamples

The only assumption being made about the starting state is that the flag *qed_ready* is asserted. An IPC proof beginning at an arbitrary, possibly unreachable, starting state will most often fail because the starting state includes an arbitrary set of value assignments to the registers of the processor.

The SQED property needs to be reformulated such that it represents more internal reachability information of the pipeline. The new formulation is shown in Fig. 3. Informally, it states: "If the CPU is in a QED-consistent state and the *qed_ready* flag is asserted, then, when the flag *qed_ready* becomes asserted again, a new QED-consistent state is reached."

Due to the fact that the initial state of the proof is left as a free input, the solver may consider an initial state in which there are some partially executed instructions in the pipeline. These pending instructions may write some new values into some registers which do not comply with QED consistency and generate a false counterexample in the end. To handle this issue, one solution is to assume a flushed pipeline in the starting state of the operation, as shown in Fig. 3.

In order to make a meaningful contribution to post-silicon validation tests, the SQED property of Fig. 3 should be proven for any possible starting state and for values of k large enough to cover QED tests of realistic size. However, increasing k drastically increases the complexity and run time of the proof. As will be reported in Sec. V, for verifying the core part of Aquarius it is not possible to obtain a proof result from the solver for $k \geq 12$ in less than 45 hours. Therefore, instead of taking this straightforward approach of incorporating a symbolic initial state into SQED, in the next section we propose a new method (S^{2}QED) which tackles this problem and aims at proving the SQED property of Fig. 2 with only a small number of unrollings that is manageable by the solver.

IV. SYMBOLIC QUICK ERROR DETECTION WITH SYMBOLIC INITIAL STATE (S^{2}QED)

A. Bug Activation and Detection

Logic bugs in a processor can be modeled in two steps: (1) a particular instruction sequence that activates the bug and (2) a particular instruction sequence that makes the bug observable in a program-visible state bit. In this bug model, the bug activation criterion can be described by the set of states that are reached by the design under verification at the end of the bug activation sequence.

In a minimal QED bug trace, there is (at least) one failing instruction in the sequence which propagates the error effect into program-visible registers; the instructions before the (first) affected instruction either contribute to the activation of the bug or they can be omitted from the trace. By this definition, the length of the error trace depends on how many instructions are needed to activate the bug. However, in S^{2}QED, if the solver is allowed to consider every possible initial state, it will be able to start from a state that implicitly represents the system after a bug activation sequence.

Hence, the error trace for every possible QED-discoverable bug can be as short as one instruction provided that there is no restriction on the initial state of the proof.

```
assume:
    at $t_{\mathrm{IF}}$:          $cpu2\_fetched\_instr()$
                    $= QED\_duplicate(\, cpu1\_fetched\_instr()\, );$
    at $t_{\mathrm{WB}} - 1$:  $qed\_consistent\_registers();$
prove:
    at $t_{\mathrm{WB}}$:         $qed\_consistent\_registers();$
```

Fig. 4. Basic S^{2}QED property

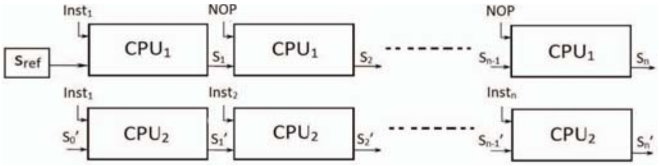

Fig. 5. S^{2}QED Verification model

B. S^{2}QED Verification model

The verification model we present in this paper is based on the idea that the original and EDDI-V-transformed sequence can be executed in parallel *on two independent instances of the CPU*. QED consistency in this case refers to a mapping between the registers of the two CPU instances. There can be an arbitrary mapping between the register names. For example, if $O = \{R^1_0, \ldots, R^1_N\}$ is the set of original registers (in instance 1) and $D = \{R^2_0, \ldots, R^2_N\}$ the duplicate register set (in instance 2), and all registers behave in the same way, one way of defining the correspondence function could be: $m(R^1_i) = R^2_{N-i}$. Note that in S^{2}QED also special registers like the PC and the status registers can be mapped between the two instances; however, usually every such special register in CPU 1 must be mapped to the same register in CPU 2. S^{2}QED can be enabled to also check the control flow if we extend the check expression *QED_consistent_registers*() by also comparing the values of the PC in the two instances after a branch instruction.

In order to simplify the presentation of the basic idea, let us for now consider a processor with a static (in-order) pipeline. For example, let us consider a classical 5-stage RISC pipeline with IF, ID, EX, MEM and WB stages. Fig. 4 shows the property that needs to be proven in order to show the absence of any QED-detectable bug. The property assumes that both CPU instances fetch the same opcode at time point t_{IF}; the instruction fetched in CPU 2 is the QED-duplicated version of the instruction fetched in CPU 1, with the same opcode but with different operands according to the register mapping m. In each CPU instance, this "instruction under verification" (IUV) passes through the pipeline and eventually writes its results to the register file in the write-back stage at time point t_{WB}.

The S^{2}QED property of Fig. 4 makes no restrictions on the initial state other than that CPU 2 executes a QED copy of the instruction in CPU 1, and that the previous instruction sequence produces a QED-consistent register file.

As an example, let us see how S^{2}QED detects the bug in Fig. 1. When checking the S^{2}QED property of Fig. 4 on the buggy pipeline the SAT solver produces one out of many valid counterexamples. One possible error trace could, for example, contain the first three instructions from this example as "original sequence" on CPU 1, and their duplicated versions on CPU 2. At time point t_{IF}, the CPU 1 fetches the "instruction under verification", `MOV R1, #3` from line 3 in Fig. 1, and CPU 2 fetches the "duplicate" instruction `MOV R17, #3` from line 6 (assuming the same register mapping as in the QED example). In the counterexample, the initial state at t_{IF} also contains the instructions from lines 1 and 2 in the EX and ID stages of CPU 1, and the instructions from lines 4 and 5 in the EX and ID stages of CPU 2. The error trace shows that at the later time point $t_{\mathrm{WB}}-1$, the instruction from line 2 writes-back into the register file of CPU 1, and the instruction from line 5 writes-back into the register file of CPU 2. At this time point, both register files are still in QED consistency with each other, as required by the assumption of the property. However, the bug has been activated in the pipeline of CPU 1 (but not in CPU 2), and the IUV from line 3 writes-back a corrupted value into the register file at t_{WB}, while the duplicate instruction writes the correct values into its register file.

Theorem 1. *The S^{2}QED property of Fig. 4 fails for all QED-EDDI-V-detectable logic bugs (cf. Def. 1), for a given register mapping m.*

Proof. Assume there is a QED-EDDI-V-detectable bug in the processor design. Then, there exists an instruction sequence 1 which starts from some QED-consistent initial state and produces a wrong result in the processor registers or memory locations, and there also exists another instruction sequence 2 with the same opcodes which produces a different result (e.g., a correct result). If we compare the register sets after each instruction of sequence 1 with the corresponding register sets of sequence 2 according to the mapping function m, as a result of Def. 1, we can identify one instruction (the IUV from above) for which the registers/memory locations are still QED-consistent before the execution of this instruction, but not QED-consistent after the execution. This is the instruction that propagates the error effect into the program-visible registers/memory locations. We call it the "observing instruction" in the following.

The property of Fig. 4 fails for a processor design containing the considered bug, because a counterexample exists that violates the property. This counterexample fetches the observing instruction at t_{IF} in CPU 1 and its non-observing duplicate in CPU 2. There are no constraints on the initial state of the IPC property other than that the instruction sequence preceding the observing instruction does not create QED-inconsistent register sets in CPU 1 and CPU 2 and that CPU 2 executes the same instruction as CPU 1, however based on different operands as given by m. The SAT solver implicitly enumerates in CPU 1 all possible instructions of the ISA, under all possible operand configurations and addressing modes. If a QED-EDDI-V-discoverable bug exists, every instruction observing the error effect of the activated bug causes the property to fail. □

Note that in S^{2}QED, actually only the instruction under verification (IUV) needs to be duplicated. There is no reason to also duplicate the opcodes of the preceding instructions. This observation allows for a modification of the property that reduces the proof complexity but does not impair the generality of the proof result: One of the two CPU instances may be restricted to start from an initial state that represents a pipeline with a predetermined instruction sequence, for example, a sequence of NOPs (flushed pipeline). Also, the instructions that follow the IUV may be constrained to be NOP instructions in that core.

Fig. 5 shows the S^{2}QED computational model. We unroll the two instances of the processor for a number of time frames. CPU 1 is constrained to start from a flushed-pipeline state S_{ref} and also fetches only NOPs in the time frames for $t > 1$. CPU 2 is unconstrained with respect to its initial state and all succeeding instructions. In this computational model, the SAT solver compares the scenario 1 where the IUV executes in a flushed-pipeline context with all possible scenarios 2 where the IUV executes in an arbitrary context including the ones where bugs are activated and propagated. (Should the bug occur in the flushed-pipeline context of CPU 1 then any other context produces a different (correct) result in CPU 2 and the bug is detected, also.) Constraining CPU 1 by fixing many inputs to constants, as shown, significantly reduces proof times.

If a design passes the S^{2}QED property then this means that there is no QED-EDDI-V-detectable logic bug (cf. Def. 1) in the design, for QED tests of arbitrary length. This is a strong statement that is very useful in post-silicon validation: It allows to conclude that any failed QED test in post-silicon validation is due to an electrical or other bug, not a logical design bug.

Furthermore, we can also make useful statements in pre-silicon verification, as already mentioned earlier. Proving the S^{2}QED property allows us to conclude for every instruction in the ISA that its execution is independent of the state of the processor when the instruction is loaded.

C. Handling Out-Of-Order Write-Back

So far in our discussion, to keep things simple, we assumed a strictly static pipeline. In practice, however, processor pipelines are more complex. Even in static pipelines, the time points of write-back are not fixed and may even happen out-of-order (as is the case for the processor in our experiments, Aquarius). Therefore, instead of assuming a QED-consistent register file at the clock cycle preceding the completion of the IUV (as in Fig. 4), we have to make an assumption that is a bit more complex. We call it the "QED-consistent pipeline" assumption. Let r^t be the content of a register r at time point t; w_r^t be the *write enable* signal for register r at time point t, i_r^t be the data input to register r at time point t, and t_{wb0} be the time point of the last register write of the instruction sequence *preceding the IUV*. O and D are the original and duplicate register sets, respectively, and m is the correspondence mapping between them, as defined earlier. With respect to the verification model shown in Fig. 5, O is the register file of CPU 2 and D is the register file of CPU 1.

$$\mathit{qed\_consistent\_pipeline}(t_0) :=$$

$$\bigwedge_{r\in O}\left(\left(r^{t_0}=[m(r)]^{t_0}\right)\vee\bigvee_{t=t_0}^{t_{\text{wb0}}} w_r^t\right) \tag{3a}$$

$$\wedge\bigwedge_{t=t_0}^{t_{\text{wb0}}}\bigwedge_{r\in O}\left(w_r^t\implies\left(i_r^t=[m(r)]^{t_0}\right)\vee\bigvee_{u=t+1}^{t_{\text{wb0}}} w_r^u\right) \tag{3b}$$

The above logic expression derives, for a practical pipeline, the conditions representing a QED-consistent register file as a result of the instructions *preceding the IUV*. The basic idea is that individual pairs of corresponding registers are QED-consistent either if the last values written into them are equal or, if nothing has been written, they have been equal at the beginning of the considered time window. The upper part (Eq. 3a) states, for every register of the original set, that the content of the register at time point t_0 is the same as in the duplicate set or else there is some pending instruction in the pipeline enabling a write into this register at some later point. The lower part (Eq. 3b) specifies that if there is a write enabled to a register at some time point t then the written value matches the corresponding value in CPU 1, or there is another write enabled to this register at some time point later than t.

Fig. 6 shows the S^{2}QED property to be proven. t_{wb1} is the write-back time point of the IUV. Compared to the SQED property of Fig. 3 with flushed-pipeline assumption, more white-box information about the microarchitecture of the processor is required to formulate the QED-consistent pipeline state. This white-box information is, however, fairly simple to obtain. It requires identifying the set of all write-enable signals for data flowing into the general-purpose register file and the other program-visible registers from within the processor pipeline. Also, the time points t_{wb1} and t_{wb0} of write-back for the IUV and the instruction preceding the IUV, respectively, need to be determined. This can be done based on few pipeline control signals, e.g., those controlling stalling. Timepoint definitions don't need to be constant and can be formulated specific to each register. This allows us to accomodate for out-of-order write-backs.

```
assume:
    at t_IF:                 cpu2_fetched_instr()
                             = QED_duplicate( cpu1_fetched_instr() );
    during [t_IF+1, t_wb1]:  cpu1_fetched_instr()= NOP;
    at t_IF:                 qed_consistent_pipeline (t_IF);
prove:
    at t_wb1:                qed_consistent_registers();
```

Fig. 6. S^{2}QED property

TABLE II
BUG DETECTION RESULTS

	Activation sequence	Property Checking [11], [12]	SQED [16]	S^{2}QED (proposed)
Bugs detectable?	short	yes	yes	yes
	long	yes	no	yes
Amount of effort		3 person months	NA	10 person hours
Run time (bugs inserted)		$<$ 1 min	$<$ 1 min	$<$ 1 min
Counterexample length (# of instructions)		1	3	1
Run time (bug-free)		25 min	Timed out	264 min

V. Experimental Results

The effectiveness of S^{2}QED is shown by verifying functional correctness w.r.t. QED-discoverable bugs of Aquarius, a 32-bit open-source static-pipeline processor with out-of-order completion which is based on the SuperH2 instruction set architecture (ISA) [18]. 18 different logic bug scenarios from [16], [17] and also the bug described in Tab. I were injected into the RTL code of Aquarius. These bug scenarios have occurred in various commercial processors and SoCs and are known to be difficult to detect and localize [16], [17]. For IPC property checking, we used the commercial tool Onespin 360 DV-Verify on an Intel Xeon E5-2660 with 32 GB of RAM, running at 2.20 GHz.

For a comparison with the original SQED approach, we instrumented the fetch unit of the Aquarius processor with a QED module, as described in [16]. (Note that in contrast to SQED the S^{2}QED method does not require any instrumentation or modification of the RTL code.) We inserted into Aquarius the logic bugs from [16], [17] that were previously detected on different hardware platforms using the original SQED technique [16]. We detected all of these bugs using S^{2}QED, with less than one minute of computation time each.

Tab. II summarizes the results. In this table, each column represents a different verification method applied to Aquarius. Aquarius was formally verified by state-of-the-art formal property checking using the industrial methodology of [11], [12]. This is represented in the first column. The second column corresponds to SQED which uses BMC as its proof method. In Tab. II, "Amount of Effort" refers to the manual work done by the verification engineer to develop the verification setup and the property. "Run time" refers to the computation spent to detect a bug or to prove its absence. In case of a bug, the length of a counterexample is also stated. Obviously, it is usually more complex for the solvers to prove the correctness of a property than to disprove it and find a counterexample.

Tab. III is dedicated to the run times and memory requirements of SQED and S^{2}QED. In the first row, we show that

TABLE III
COMPARING SQED (FOR REALISTIC QED TEST LENGTHS) WITH S^{2}QED (FOR ARBITRARY TEST LENGTH)

Property	k	**Run time**	**Memory**
SQED property with symbolic initial state (Fig. 3)	12	>45 h	2566 MB
SQED with specific initial state [16]	10	4:49	1325 MB
SQED with specific initial state [16]	12	1:46:33	1482 MB
SQED with specific initial state [16]	14	42:14:51	1878 MB
S^{2}QED	7	4:24:18	3809 MB

the straightforward approach of integrating a symbolic initial state, as in Fig. 3, is not tractable. Then, we compare our implementation of SQED with S^{2}QED. For a comparison between SQED and S^{2}QED it should be noted that the S^{2}QED computational model represents the duplicated instruction sequence in parallel (cf. Fig. 5) while conventional SQED, as a result of its QED module, represents the same sequences sequentially one after the other. Therefore, in terms of the model size, $k = 7$ unrollings in S^{2}QED correspond to $k = 14$ unrollings in SQED. For S^{2}QED a time window of $k = 7$ clock cycles was chosen because this is sufficient to formally prove the absence of QED-discoverable bugs, since every instruction can finish execution in this time interval. A larger window is not needed and would not strengthen the proof in any way. In SQED, however, the strength of the method depends on the length of the time window. We therefore examine SQED with a different number of unrollings.

Based on the above two tables the following observations can be made:

Observation 1: S^{2}QED can detect all bugs in reasonable time that are discoverable by the original SQED method. It can also discover bugs with long error detection latencies, e.g., the bug described in Tab. I, which needs an activation sequence of 14 instructions, with a bug trace of only one instruction. Given that the original SQED cannot detect this class of bugs, this feature significantly improves the verification power of S^{2}QED. Since the length of the bug trace strongly affects the complexity of the proof, being able to find difficult bugs with very short bug traces makes the method more scalable for larger designs.

Observation 2: As described in previous sections, S^{2}QED makes a valuable contribution to pre-silicon verification by proving that each instruction behaves independently of its context in the program. Bugs escaping this check will occur equally in every context and therefore, as also noted in [9], can be expected to be easily detected even by simulation-based methods. Since no golden model of the ISA needs to be specified, S^{2}QED can obtain this important verification result with substantially less manual effort compared to classical processor verification by property checking (10 person hours as opposed to 3 person months).

Observation 3: S^{2}QED requires no modification in the RTL code of the design and it has no restriction on the instructions that it can consider, unlike the original SQED which is not able to consider certain instructions that use operands other than general purpose registers, e.g., status registers. The verification setup can be reused for patched or customized versions of the processor core as long as the program-visible registers in the design are not changed.

VI. CONCLUSION

The paper presented S^{2}QED, a formal method to find all QED-discoverable logic bugs inside processor cores during pre-silicon verification. This is a major improvement over the previous SQED method. As opposed to SQED, S^{2}QED also makes a well-defined contribution to pre-silicon verification: it proves that the result of every instruction execution is independent of its context in the program. The method requires considerably less manual effort compared to previous methods of property checking.

With our notion of a QED-consistent pipeline, as formulated in Eq. 3 of Sec. IV-C, the proposed approach proved useful for processors with out-of-order write-back. However, our current approach will face its limits for fully dynamic processors with out-of-order execution and reorder buffers, due to their large sequential depth. Improving tractablity for such cases, therefore, is subject to our future work.

REFERENCES

[1] H. D. Foster, “Trends in functional verification: A 2014 industry study,” in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[2] A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv, “Threadmill: A post-silicon exerciser for multi-threaded processors,” in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2011, pp. 860–865.

[3] S. Mitra, S. A. Seshia, and N. Nicolici, “Post-silicon validation opportunities, challenges and recent advances,” in *Design Automation Conference*, June 2010, pp. 12–17.

[4] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessors control,” in *Proc. International Conference Computer Aided Verification (CAV)*, D. L. Dill, Ed., vol. 818. Standford, California, USA: Springer-Verlag, 1994, pp. 68–80. [Online]. Available: citeseer.ist.psu.edu/burch94automatic.html

[5] R. B. Jones, D. L. Dill, and J. R. Burch, “Efficient validity checking for processor verification,” in *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, Nov 1995, pp. 2–6.

[6] W. Damm, A. Pnueli, and S. Ruah, “Herbrand automata for hardware verification,” in *Proceedings of the 9th International Conference on Concurrency Theory*, ser. CONCUR ’98. London, UK, UK: Springer-Verlag, 1998, pp. 67–83. [Online]. Available: http://dl.acm.org/citation.cfm?id=646733.701323

[7] S. Berezin, A. Biere, E. Clarke, and Y. Zhu, *Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-Order Processor Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 369–386.

[8] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal, “Bdd based procedures for a theory of equality with uninterpreted functions,” in *Proceedings of the 10th International Conference on Computer Aided Verification*, ser. CAV ’98. London, UK, UK: Springer-Verlag, 1998, pp. 244–255. [Online]. Available: http://dl.acm.org/citation.cfm?id=647767.733760

[9] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, *End-to-End Verification of Processors with ISA-Formal*. Cham: Springer International Publishing, 2016, pp. 42–58.

[10] M. R. Prasad, A. Biere, and A. Gupta, “A survey of recent advances in SAT-based formal verification,” *International Journal on Software Tools for Technology*, vol. 7, no. 2, pp. 156–173, apr 2005.

[11] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, “Complete formal verification of TriCore2 and other processors,” in *Design & Verification Conference & Exhibition (DVCon)*, 2007.

[12] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, “Unbounded protocol compliance verification using interval property checking with invariants,” *IEEE Transactions on Computer-Aided Design*, vol. 27, no. 11, pp. 2068–2082, November 2008.

[13] J. Urdahl, D. Stoffel, J. Bormann, M. Wedler, and W. Kunz, “Path predicate abstraction by complete interval property checking,” in *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2010, pp. 207–215.

[14] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS ’99. London, UK, UK: Springer-Verlag, 1999, pp. 193–207.

[15] D. Lin, E. Singh, C. Barrett, and S. Mitra, “A structured approach to post-silicon validation and debug using symbolic quick error detection,” in *Proc. IEEE Intl. Test Conf. (ITC)*, Oct 2015, pp. 1–10.

[16] E. Singh, D. Lin, C. Barrett, and S. Mitra, “Logic bug detection and localization using symbolic quick error detection,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, (to appear).

[17] D. Lin, T. Hong, Y. Li, E. S, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, “Effective post-silicon validation of system-on-chips using quick error detection,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1573–1590, Oct 2014.

[18] T. Aitch, “Aquarius: a pipelined RISC CPU,” 2003. [Online]. Available: http://opencores.org/project,aquarius