Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED

(Invited Paper)

Florian Lonsing, Karthik Ganesan, Makai Mann, Srinivasa Shashank Nuthakki,
Eshan Singh, Mario Srouji, Yahan Yang, Subhasish Mitra, and Clark Barrett
Computer Science Department, Stanford University, Stanford, CA 94305, USA
E-mail: {lonsing, karthik3, makaim, nuthakki, esingh, msrouji, yangy96, subh, barrett}@stanford.edu

***Abstract*—As designs grow in size and complexity, design verification becomes one of the most difficult and costly tasks facing design teams. Formal verification techniques offer great promise because of their ability to exhaustively explore design behaviors. However, formal techniques also have a reputation for being labor-intensive and limited to small blocks. Is there any hope for successful application of formal techniques at design scale? We answer this question affirmatively by digging deeper to understand what the real technological issues and opportunities are. First, we look at satisfiability solvers, the engines underlying formal techniques such as model checking. Given the recent innovations in satisfiability solving, we argue that there are many reasons to be optimistic that formal techniques will scale to designs of practical interest. We use our CoSA model checker as a demonstration platform to illustrate how advances in solvers can improve scalability. However, even if solvers become blazingly fast, applying them well is still labor-intensive. This is because formal tools are only as useful as the properties they are given to prove, which traditionally have required great effort to develop. Symbolic quick error detection (SQED) addresses this issue by using a single, universal property that checks designs automatically. We demonstrate how SQED can automatically find logic bugs in a variety of designs and report on bugs found and efficiency gains realized in academic and industry designs. We also present a generator for an improved SQED module that further reduces the amount of manual effort that has to be spent by the designer.**

***Index Terms*—symbolic quick error detection, bounded model checking, SAT solving, verification, validation.**

I. Introduction

Pre-silicon verification accounts for a significant fraction of overall design effort [1]. Even so, conventional pre-silicon verification techniques are not thorough enough to find corner-case logic bugs, especially in large or complicated designs. These challenges are further magnified by the slowdown of the classical silicon CMOS (Dennard) scaling [2], as integrated circuits (ICs) increase design complexity tremendously to meet speed and energy targets.

As a result of these challenges, critical logic design bugs frequently escape pre-silicon verification and are detected only after ICs are manufactured, during post-silicon validation or during system operation. Bugs found in post-silicon validation can be extremely expensive and difficult to localize and fix. Furthermore, bugs that are not discovered until system operation can have disastrous consequences, especially in safety-critical domains such as automotive applications. Even when bugs are detected early, root-causing and fixing them can take weeks or months of effort. This is because bugs may take millions or even billions of clock cycles to detect using traditional verification techniques. It is then notoriously hard to identify what went wrong and when.

A promising approach for addressing these challenges is to employ formal techniques such as model checking [3]. Model checking constructs a mathematical representation of the system and then attempts to formally prove that this representation has certain desired properties. Such a representation can be constructed in a straightforward way (e.g., from a Verilog model of the system). Formal techniques are valuable because they are *exhaustive*: proving a property guarantees that it holds for all possible executions of the system. Moreover, when the proof fails, a *counterexample* or *bug trace* is produced (a sequence of instructions sufficient to trigger and detect a bug). However, formal techniques have traditionally suffered from two main drawbacks. First, traditional application of formal techniques requires substantial experience to write meaningful properties. Finding the right set of properties to represent an informal, high-level design specification is challenging [4]. Consequently, the quality of the formal verification process depends on the expertise of the verification engineers. The second challenge is scalability. Even with the right set of properties, formal techniques have traditionally been unable to scale to a full chip or SoC, limiting their applicability to small blocks.

In this paper, we present evidence that tremendous progress is being made on both of these fronts. For scalability, we review recent progress in automated reasoning tools such as Boolean satisfiability (SAT) and satisfiability modulo theories (SMT) solvers. These tools are the main back-end engines used by modern formal tools [5]–[10]. Improvements to such tools is an ongoing research effort which has seen dramatic advances recently, greatly enhancing the scalability of formal tools.

However, such technological progress alone will not lead to a more widespread application of formal techniques. The manual effort required to write properties must also be

This work was partially supported by a National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported by the Defense Advanced Research Projects Agency, grants FA8650-18-1-7818 and FA8650-18-2-7854.

reduced. To this end, we present *symbolic quick error detection (SQED)*, an easy-to-apply formal technique that has been successfully used for pre- and post-silicon design verification and validation [11], [12]. SQED utilizes model checking to prove that any instruction sequence up to a certain bound produces a correct result. SQED leverages the idea of design self-consistency to formulate a single universal property that is inherently design-independent. Therefore, SQED does not require any manual property formulation.

The rest of the paper is organized as follows. We first provide an overview of SAT, SMT, and model checking and explain their relevance for hardware verification. Then we introduce SQED together with a new generator-based approach, focused on the verification of processor cores. As a formal technique that is complementary to SQED, we present single-instruction checking. It detects bugs resulting from incorrect implementations of single instructions in a processor design. Finally, we conclude with experimental results that demonstrate: (i) the impact of SAT and SMT solver improvements on the performance of formal tools generally, and SQED in particular; and (ii) the ability of SQED to automatically find real bugs in a large processor design, specifically the RIDECORE [13] design, an open-source RISC-V processor core.

II. Tools for Formal Hardware Verification

A. SAT Solving and Satisfiability Modulo Theories

The *satisfiability problem of propositional logic (SAT)* is the canonical NP-complete problem [14]. In the 1970s and 1980s, SAT-related research focused on theory. The advent of powerful SAT solving algorithms and their efficient implementation in *SAT solvers* in the mid 1990s led to a surge of practical SAT applications, a trend dubbed the "SAT revolution" [15]–[17].

In formal verification, many problems can be encoded as a SAT problem and solved using a SAT solver. For example, digital circuits can be modeled as state transition systems, which can be encoded as propositional formulas. To prove a property of the system, a SAT problem is constructed by combining these encodings with an encoding of the property.

Despite the NP-completeness of SAT, modern SAT solvers such as CryptoMiniSat,[1] CaDiCaL [18], or Lingeling [19] routinely solve problems with millions of variables. On practically relevant instances, SAT solvers rarely exhibit worst-case exponential runtime. This is due to structure in instances that can be exploited by solvers. Moreover, modern solver implementations are based on sophisticated engineering and employ advanced heuristics and techniques that greatly improve upon the original, basic backtracking search approaches.

Satisfiability modulo theories (SMT) [20] is a generalization of propositional satisfiability to decidable fragments of first-order logic. Fragments of particular relevance to formal hardware verification are the theories of bit-vectors and arrays. Compared to propositional logic, where individual bits are modeled using propositional variables, the theories of bit-vectors and arrays make it possible to model memories and words, respectively, in a more natural and concise way. The concise representation enables reasoning at the word-level rather than the bit-level. In general, word-level reasoning is more powerful than bit-level reasoning. Modern SMT solvers (e.g., Boolector [21], CVC4 [22], Yices2 [23], Z3 [24]) combine word-level reasoning with low-level SAT reasoning to solve problems that are beyond the reach of SAT alone.

[1] https://github.com/msoos/cryptominisat

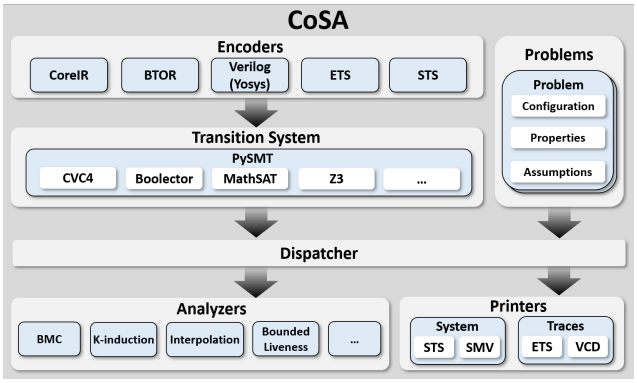

Fig. 1. CoSA Architecture.

B. Model Checking

The increased capacity of automated reasoning in SAT and SMT solvers has been driving formal techniques like model checking [3]. Two main kinds of model checking are used. *Unbounded* model checking is exhaustive: if a property is proved to hold in a system, then the property holds for all possible executions of the systems. However, unbounded model checking may be infeasible for large systems.

Bounded model checking (BMC) [25] guarantees exhaustiveness with respect to executions of a certain length. In practice, the bound on the length of the execution is incrementally increased up to a computational or user-defined limit, or until a counterexample to the property is found. BMC always produces the shortest possible counterexample, if one exists.

BMC is implemented using efficient SAT and SMT solvers. Thus, checking whether there exists a counterexample to a given property is reduced to solving a SAT or SMT problem. To do this, the model of the system is *unrolled* over time by replicating the circuit for each time-step. A crucial feature of SAT and SMT solvers for efficient BMC is *incrementality*. Incremental solvers leverage effort spent solving the BMC problem at bound k when solving the BMC problem at bound $k+1$.

C. The Model Checker CoSA

The CoreIR Symbolic Analyzer (CoSA) [26] is an open-source, SMT-based hardware model checker. CoSA leverages state-of-the-art satisfiability modulo theories (SMT) solvers and incorporates word-level design structure to improve verification performance. Fig. 1 depicts the high-level architecture. CoSA supports several standard input formats, in addition to the explicit transition system (ETS) and symbolic transition system (STS) formats. ETS and STS inputs provide a convenient method for adding temporal constraints on system executions,

for example providing a reset sequence. The SMT encoding for BMC implemented in CoSA is a straightforward extension of the respective SAT encoding to word-level variables.

As demonstrated by competitions [27], [28], SAT and SMT solvers are improving regularly. CoSA is built modularly to easily incorporate the latest solver enhancements. The transition system representation utilizes PySMT [29] which allows modular swapping of the underlying SMT solver. Moreover, many SMT solvers also have a modular interface to SAT solvers. Thus, automated reasoning improvements anywhere in the toolchain can quickly propagate through the system by updating or swapping solvers, enabling faster verification times.

III. Symbolic Quick Error Detection

A. Overview

In the following, we present the basic concepts behind symbolic quick error detection (SQED). We start with an explanation of its predecessor, quick error detection.

1) Quick Error Detection: Quick error detection (QED) is a testing technique that takes an existing test consisting of a sequence of instructions and automatically transforms it into a new test using various *QED transformations* [30]. The purpose of these transformations is to improve coverage and reduce error detection latency (defined as the number of cycles between when an error occurs and becomes observable). An error is observable once the architectural state is wrong, for example, when an incorrect value is stored in a general-purpose register. One of the most powerful QED transformations is *error detection using duplicated instructions for validation (EDDI-V)* [30]. The EDDI-V QED transformation uses shadow registers and memory to duplicate the instructions in an existing instruction sequence. More specifically, the registers and memory space of the design under test (DUT) are divided into two halves so that each register or memory location in one half is mapped to exactly one register or memory location in the other half by means of a bijective mapping. Each half is referred to exclusively by the original or duplicated instructions, respectively. The EDDI-V QED transformation copies each original instruction to a duplicate one, with the change that the register and memory locations are mapped to their corresponding values for duplicate instructions. In a *QED test*, both the original and the duplicated instruction sequences are executed from a system state where original and duplicate registers and memory locations hold the same values. Such a state is called *QED-consistent*. Duplicated instructions execute in the same relative order as the original ones but may be interleaved. If the QED test produces a state where the values held by original and duplicate registers or memory locations do not match, then the QED test constitutes a bug trace.

2) Symbolic Quick Error Detection: Symbolic quick error detection (SQED) [11], [12] combines QED transformations with *bounded model checking (BMC)* [25]. The model checker systematically enumerates all possible instruction sequences of increasing length in a *symbolic* way. QED transformations are then applied to the enumerated instruction sequences. Original and duplicated instruction sequences are symbolically executed on a model of the DUT represented in some hardware description language. The use of symbolic methods is in contrast to traditional QED, which applies QED transformations to a given, concrete instruction sequence.

```
INPUT: [31:0] instruction, clock
assign opcode = instruction[6:0];
assign funct3 = instruction[14:12];
assign funct7 = instruction[31:25];
assign ADD =
  (funct3 == 3'b000) && (opcode == 7'b0110011) &&
  (funct7 == 7'b0000000);
// add opcode constraints for all instructions in ISA
...
always @(posedge clock) begin
    assume property (ADD ||...||...);
end
```

Fig. 2. Opcode constraint example: 32-bit register-type `ADD` of RISC-V ISA.

Compared to more traditional uses of model checking, the main advantage of SQED is that the user does not need to formulate the properties to be checked. Instead, SQED uses a single *universal property*: if a QED test is run from a QED-consistent state, then the resulting state must also be QED-consistent. For example, for a processor core with 32 general-purpose registers, a state is QED-consistent if the following property holds:

$$\bigwedge_{i:=0}^{15} regs[i] = regs[i+16]$$

where $regs$ denotes the register file. We assume that registers zero to 15 are designated original and 16 to 32 are duplicate, and register i is mapped to register $i + 16$. SQED also automatically finds the *shortest possible bug trace*, if a bug trace exists, or proves the absence of bug traces with respect to instruction sequences up to the considered length.

To prevent spurious counterexamples to the universal property, the model checker must only select instructions that are part of the ISA implemented by the DUT. This is achieved by providing the model checker with constraints that express the opcodes of all the instructions in the ISA. Fig. 2 shows the opcode constraint of the 32-bit register-type `ADD` instruction of the RISC-V ISA. The constraints are specified to the model checker as a simple, disjunctive SystemVerilog property.

To implement SQED, a special *QED module* is integrated with the DUT. It is used only for pre-silicon verification and is not added to the manufactured integrated circuit. The QED module takes as input a stream of original instructions and produces an output stream of instructions that corresponds to a QED test based on the original instructions. The input stream is allowed to be symbolic (selected by the model checker), and the output stream is fed into the DUT. The QED module implements a *QED-ready* signal, which during the run of a QED test asserts the points in time when the numbers of original and duplicate instructions executed are the same. At such points in time, the processor state should be QED-consistent, and hence the model checker checks whether the universal property holds. To be valid, the model checker must be started from

```
INPUT:  enable, next_instr, fetch_next, original
OUTPUT: instr_out, instr_valid

//begin initialization
queue := 0, head_instr := 0;
//end initialization

insert_valid := fetch_next & original & ~queue.full();
delete_valid := fetch_next & ~original & ~queue.empty();
instr_valid  := insert_valid | delete_valid;

if insert_valid then
  queue.enqueue(next_instr);
else if delete_valid then
  head_instr := queue.dequeue();
endif

dup_instr := create_duplicate_version (head_instr);
instr_out := (enable & ~original) ? dup_instr : next_instr;
```

Fig. 3. Pseudocode for the new QED module.

```
INPUT: write_valid, write_address
OUTPUT: qed_ready

//begin initialization
qed_ready := false, cnt_orig := 0, cnt_dup := 0;
//end initialization

is_original := is_write_to_original_space (write_address);

if write_valid then
  if is_original then
    // increment number of committed original instructions
    cnt_orig++;
  else
    // increment number of committed duplicate instructions
    cnt_dup++;
  endif
endif

qed_ready := (cnt_orig == cnt_dup) ? true : false;
```

Fig. 4. Pseudocode for the QED-ready logic of the new QED module.

a QED-consistent state. Such a state can be a reset state, for example, or a QED-consistent state obtained from simulation.

B. New QED Module

In the following, we present a new QED module [31] that improves upon previous implementations [12]. The main improvement is that this module allows original and duplicate instructions to be interleaved (previous implementations simply ran all original instructions followed by all duplicate instructions). As with previous QED modules, it is designed to be connected to the instruction fetch stage of a processor core. The pseudocode of the new QED module is given in Fig. 3.

We first explain the QED module inputs and outputs. The `enable` signal disables the QED module if set to zero, resulting in the execution of original instructions only. Signal `next_instr` is the next original instruction to be executed; `fetch_next` indicates whether the core is ready to receive an instruction, i.e., the fetch stage is not stalled. Signal `original` indicates whether to execute an original or its corresponding duplicate instruction. Outputs of the QED module are signal `instr_valid`, which indicates whether the output instruction should be considered valid, and the actual original or duplicate instruction `instr_out` to be executed.

To apply QED transformations, the QED module maintains a queue of original instructions which have not yet been duplicated. When `original` is high, the QED module triggers the execution of the current original instruction (`next_instr`) after enqueueing it for later duplication. When `original` is low, an instruction is removed from the queue (`head_instr`), then duplicated (`dup_instr`) and sent for execution. Notice that both `next_instr` and `original` are inputs, thus the model checker can choose their values freely (as long as `next_instr` satisfies the ISA constraints, cf. Fig. 2). Signals `insert_valid` and `delete_valid` provide additional conditions on enqueueing or dequeueing instructions. Pipeline stalls invalidate the output instruction of the QED module (signals `fetch_next` and `instr_valid`).

In SQED, the model checker checks the universal property each time the same number of original and duplicate instructions have committed. Checking the property prematurely might produce spurious counterexamples. Fig. 4 shows the pseudocode of the QED-ready logic. The output signal `qed_ready` goes high whenever the same number of original and duplicate instructions have committed. The implementation of the QED-ready logic is design-dependent, as it has to be customized to different pipeline implementations. Signal `write_address` is the address of the data to be written and signal `write_valid` tells whether the input data is valid. Signal `is_original` indicates whether the data is written to an original or duplicate register or memory location, which is determined by applying function `is_write_to_original_space` to `write_address`. For simplicity, we assume that at most one instruction commits per cycle. For superscalar processors that can commit multiple instructions in the same cycle, we track corresponding pairs of `write_valid` and `write_address` signals and maintain a separate `is_original` signal for each executed instruction.

The QED-ready logic as shown in Fig. 4 is only applicable to single processor cores. For multi-core systems, original and duplicate commits across all cores must be considered. This can be challenging if the cores operate with a shared address space. In superscalar processors with explicit register renaming such as MIPS 10000 [32] and ARMs Cortex-A15 [33], logical addresses must be mapped to physical ones via the register mapping table before comparing values held by original and duplicate registers and memory locations. The RISC-V cores we consider in our experiments (Section IV) are single cores and do not apply register renaming.

C. An SQED Generator

Although the implementation of the QED module (Fig. 3) must be adapted to a given DUT with respect to both the QED-ready logic (Fig. 4) and the opcode constraints of the instructions in the ISA (Fig. 2), its basic functionality, that is, duplicating instructions, is design-independent. We exploit this design-independence in a generator-based workflow implemented in Python for processor cores. The generator takes

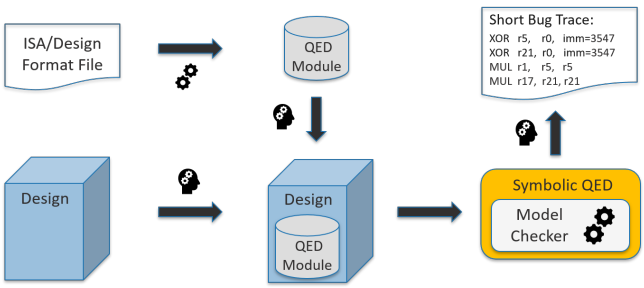

Fig. 5. Workflow of generator-based SQED.

```
SECTIONS = ISA QEDCONSTRAINTS REGISTERS BITFIELDS...

_ISA                                _BITFIELDS
num_registers = 32                  funct7 = 31 25
instruction_length = 32             funct3 = 14 12
                                    rd = 11 7
_QEDCONSTRAINTS                     rs1 = 19 15
half_registers = 1                  rs2 = 24 20
                                    opcode = 6 0
_REGISTERS                          ...
rd rs1 rs2

_R
ADD
funct3 = 000 funct7 = 0000000 opcode = 0110011
...
```

Fig. 6. RISC-V format file example (excerpt).

a structured description of the core and produces a QED module. Fig. 5 illustrates our workflow. We will use RIDECORE [13], a RISC-V core, as a running example. A walkthrough for this example is also available in a GitHub repository [34].

The format file (Fig. 6) specifies parameters of the DUT and the SQED workflow such as the number of registers, the length of an instruction, and the mapping of original to duplicate registers (`half_registers = 1`) by dividing the register file in two halves. Section headers like "`_ISA`" are prefixed with an underscore. Symbolic names are introduced for the register operands of instructions (section `_REGISTERS`). Section `_BITFIELDS` lists decoding information needed to retrieve the register operands and opcode from a given instruction by extracting the respective bits.

In the RISC-V ISA the opcode is given by the bits at index zero up to six in the instruction encoding (`opcode = 6 0`). We list the opcodes for all instructions in the ISA grouped by types such as register- (`_R`) or immediate-type. The format file for RIDECORE [13] has only about 200 lines of text. Writing the file is straightforward, as most of the necessary information should be readily available in the ISA documentation.

Given the format file as input, the generator automatically generates the Verilog implementation of the QED module. For RIDECORE, the Verilog implementation consists of about 450 lines of code. In particular, the opcode constraints that are crucial for the SQED workflow, as they prevent spurious counterexamples, are generated automatically (cf. generated opcode constraints for `ADD` in Fig. 2 and the respective opcode information in the format file in Fig. 6).

Given the generated QED module, the next step is to wire it up to the design, connecting it to the instruction fetch stage. This requires some manual effort. However, the QED module generator has been designed to make this as straightforward as possible by providing a general and well-documented interface for the generated QED module. The goal is for a designer to be able to do it themselves, even if they have no experience with formal verification. In future work, we aim to further simplify the wiring-up process by ensuring that the generated module interface works cleanly across a variety of ISA's and designs.

Applying SQED to the design with the wired-up QED module is completely automatic. If there is a bug in the design then SQED will produce a shortest bug trace that is sufficient to reproduce the bug. For example, our model checker CoSA produces a value change dump (VCD) file from which the actual bug trace can be extracted.

D. Single-Instruction Checking

SQED is effective in producing short bug traces of bugs that are otherwise difficult to find [11], [31]. However, it cannot detect bugs that are triggered by a single faulty instruction. Such *single-instruction bugs* affect original and duplicate instructions in a QED test in the same way. Hence, original and duplicate registers always hold the same value, thus producing QED-consistent states. It is thus necessary to use some complementary technique to detect single-instruction bugs. Several approaches are possible. These bugs are relatively easy to find and reproduce, so conventional testing via simulation or emulation can be quite effective. Formal techniques can also work well, and several have been proposed (e.g., [10], [31]).

Here, we outline a formal approach to single-instruction checking using the RISC-V cores Vscale [35] and RIDECORE [13] as examples (for the latter, a demo is available online [36]).[2] The ideas here are not new; the goal is rather to show how a simple formal technique can be easily integrated with SQED. For each instruction, we formulate a correctness property, which is then checked using a model checker. Once a property has been formulated for one instruction, it can be adapted to similar instructions with relatively little effort. Moreover, the correctness properties for instructions from a given ISA lend themselves to being easily portable to other DUTs with the same ISA, since these instructions have the same semantics regardless of the implementation.

Fig. 7 shows the pseudocode of the correctness properties of the 32-bit register-type `ADD` instruction for RIDECORE and Vscale, respectively. To prevent spurious counterexamples, it is necessary to include assumptions reflecting design-specific implementation details such as the pipeline. We maintain a state counter (`stcnt`) that is incremented in each clock cycle to be able to refer to the data values produced in particular pipeline stages. Along with the data that is necessary for setting up the correctness properties, the state counter is part of a helper state transition system (Fig. 7, bottom). A next-state function `next`

[2]More complicated cores, including for example support for floating point instructions are directions for future work, cf. [37].

```
assumptions: reset = 1;
             stcnt = 0 | stcnt >= 2 -> instr = NOP;
             stcnt = 1 -> instr = ADD & rd != 0;
property:    stcnt = 7 -> val1 + val2 = regs[rd_copy];
```

Correctness property (RIDECORE).

```
assumptions: reset = 0;
             stcnt = 0 | stcnt >= 2 -> instr = NOP;
             stcnt = 1 -> instr = ADD & rd != 0;
property:    stcnt = 4 -> val1 + val2 = regs[rd_copy];
```

Correctness property (Vscale).

```
next(instr)    = stcnt = 0 | stcnt >= 2 ? NOP : instr;
next(stcnt)    = stcnt++;
next(val1)     = stcnt = 1 ? regs[rs1] : val1;
next(val2)     = stcnt = 1 ? regs[rs2] : val2;
next(rd_copy)  = stcnt = 1 ? rd : rd_copy;
```

Helper state transition system.

Fig. 7. Single-instruction checking for RIDECORE and Vscale.

defines how the data values evolve from one clock cycle to the next. The helper transition system is added to the model of the DUT via synchronous composition.

We make sure that the core is not in reset mode (`reset = 1` and `reset = 0`, respectively), where the reset signal is active-low for RIDECORE and active-high for Vscale. We run all single-instruction checks from a predefined reset state.[3] Instruction fetching is complete in cycle one (`stcnt = 1 -> instr = ADD`), and the instruction (`instr`) is restricted to `NOP` in cycle zero and cycles greater than or equal to two. This is necessary to make sure that current operand values do not interfere with operands of instructions issued subsequently.

We extract the values held by the two source registers `rs1` and `rs2` in clock cycle one and store them in variables `val1` and `val2` and make a copy `rd_copy` of the destination register `rd`. Register zero is hard-wired to value zero and hence cannot be used as a destination register (`rd != 0`). Since RIDECORE and Vscale have pipelines with six and three stages, respectively, we check the actual correctness property in cycles seven and four, respectively.

The example in Fig. 7 shows the correctness property for the register-type `ADD` instruction. The properties for the other ALU instructions can be obtained easily by replacing addition by the respective operator in the expression that computes the result (`val1 + val2`) and adapting the check for the instruction opcode (`instr = ADD`). Immediate-type instructions can be handled analogously. For checking the instruction opcodes, we can re-use the constraints from SQED (Fig. 2).

Note that the main difference in the correctness properties for RIDECORE and Vscale is in the respective pipeline depths. Consequently, once the correctness properties are formulated for one design, it is easy to port them to the other design by making the pipeline depth a parameter. In general, the correctness properties could also be generated largely automatically using our generator (Fig. 5), provided that the necessary design-dependent parameters are listed in the format file (Fig. 6).

[3]Note that if a single instruction fails only in a non-reset state, then it is not a deterministically failing instruction, in which case SQED can find the bug.

TABLE I
PREVIOUSLY UNKNOWN BUGS IN RIDECORE.

Bug Activation	Bug Effect	Runtime (sec)
All but one (buggy) RS-m entries occupied, MULH instruction assigned to vacant entry.	First source operand of MULH instruction corrupted.	63
Same as above.	Second source operand of MULH instruction corrupted.	69
Same as above, but with a MULHU instruction.	Result of MULHU instruction corrupted.	93

IV. Case Studies and Experimental Results

In the following, we report on experimental results with our implementations of SQED and single-instruction checking. Experiments reported in Sec. IV-A were carried out on an AMD Opteron 6348 with 128GB of RAM, and the experiments in the other sections on a system with two eight-core AMD Ryzen 7 2700 processors and 32GB of RAM. We used Questa version 10.5c from Mentor Graphics and CoSA as bounded model checking engines for SQED.

A. New QED Module

We demonstrate the effectiveness of our new QED module [31] (Sec. III-B) on the two open-source RISC-V processor cores, Vscale [35], an in-order core targeting embedded applications, and RIDECORE [13], an out-of-order superscalar core with a two-way pipeline, 64 maximum instructions in-flight, two ALUs, one multiplier, and one load/store unit for high-performance applications. All experiments reported in this section are based on running the instruction sequences in SQED starting from a reset state and were obtained using Questa.

Using SQED with our new QED module, we found three previously unknown logic bugs in the multiplier reservation station (RS-m) of RIDECORE, all of which were confirmed by RIDECORE designers [38], as shown in Table I. Importantly, these bugs were detected only because our new QED module allows original and duplicate instructions to be interleaved. To detect the bugs, original and duplicate multiplication instructions must execute in subsequent clock cycles. The previous QED module [12] could not trigger that condition.

Using the same setup for SQED, we found two previously unknown bugs in Vscale (Table II) in less than 40 seconds, which were also confirmed by designers. These bugs are due to errors in the Vscale implementation of the RISC-V privileged ISA affecting specific control status registers (CSRs). Importantly, Vscale does not implement shadow registers for CSRs. Therefore, to implement instruction duplication we store the values of duplicate CSRs in data memory. The first bug in Table II occurs because of incorrect design of the MIP register interrupt bit logic. After this bug was fixed, we found a second bug affecting the MSTATUS register.

B. Single Instruction Checking

Table III shows experimental results for single-instruction checking applied to RIDECORE and Vscale using CoSA. The

TABLE II
PREVIOUSLY UNKNOWN BUGS IN VSCALE.

Bug Activation	Bug Effect	Runtime (sec)
Value 1 written to specific bit positions in the CSR MIP.	MTIMECMP register corrupted, causes repeated interrupts.	2
Any value with lower two bits 01 or 10 written to CSR MSTATUS.	Unspecified privilege level entered, MEPC register corrupted.	33

TABLE III
SINGLE-INSTRUCTION CHECKING FOR RIDECORE (R) AND VSCALE (V).

Instructions Checked	Runtime/Check (sec)	
	R	*V*
All instructions except MUL	40	3
All instructions with restricted MUL	40	3
MUL with injected bug	40	14

pipeline depths of the designs (six stages in RIDECORE and three stages in Vscale) determine the necessary number of unrollings in BMC, i.e., seven for RIDECORE and four for Vscale. Except for multiplication (MUL), we successfully verified the correctness of all instructions within 40s per instruction check for RIDECORE and 3s for Vscale.

Checking multiplication circuits is a known hard problem for formal techniques. One approximation that can detect many bugs is to fix one of the operands.[4] Another is to limit the bit-width of operands;[5] and another is to debug multiplication instructions using extensive tests. There are also specific formal techniques that target multiplication [39]–[41]. Integrating such techniques is a topic for future investigation.

C. Applying SQED to RIDECORE

We applied SQED using the new QED module [31] (Section III-B) to RIDECORE [13], [34]. Table IV shows results we obtained using our model checker CoSA. In the experiments, we consider the third bug in Table I as a case study. To study the impact of improvements in SAT and SMT solvers, we show results using two versions of the SMT solver Boolector [21], a base version and an improved one, which handles incremental SAT problems more efficiently.[6] We also evaluate the impact of two different SAT solvers, the older Lingeling [19] solver, and the new CaDiCaL [18] solver (which requires the improved Boolector version). We consider variations of RIDECORE obtained by design changes that preserve bug observability.

Columns **T (b)** and **T (c)** in Table IV show the time taken by CoSA (in seconds) to find the bug and, after fixing the bug, to prove the correctness of the design up to the last BMC unrolling. Column **k** shows the number of BMC unrollings.

[4]The second row of Table III shows the result of restricting the multiplication instruction to check the simpler property where one of the operands is set to constant zero or one. That restricted MUL instruction could be checked within the same time as the other instructions and could easily detect many bugs that we injected in the multipliers.

[5]We reduced the bit-width and were able to verify a 13-bit multiplier in RIDECORE and a 15-bit multiplier in Vscale within a time out of two hours.

[6]The improved (base) version of Boolector we used is located on the *smtcomp19* (*master*) branch in the GitHub repository of Boolector at https://github.com/Boolector/boolector and has commit ID 176eff (1971ce).

Each line in the table shows the results of one configuration of RIDECORE. In the default configuration, we applied only the minimal set of changes to the design needed to wire up the QED module. The bug is detected with a BMC unrolling depth of 23. Before executing any QED tests in SQED, we run a reset sequence and then start SQED from the reset state of the design. The reset sequence accounts for three of the 23 unrolling steps, and the remaining 20 steps are required for the instructions in the bug trace to execute and commit.

We can reduce the number of steps required to find the bug from 20 to 10, resulting in a total depth of 13 including the reset sequence, by removing behavior triggered on negative clock edges. This is possible in RIDECORE by disabling branch prediction and removing the branch target buffer. This does not affect the bug in question, so we also included this variation in our experiments. Taking CaDiCaL as an example, removing negative-edge clock behaviors resulted in a decrease in both the time required to find the bug (226s vs. 98s) and the time to prove correctness up to the respective BMC depth after bug fixing (697s vs. 623s). As another design variation, we ran the experiment with a smaller data memory (shrinking the size from 2048 entries to 32 entries). This further improved the model checking times (98s vs. 86s and 623s vs. 568s). These design variations illustrate that reducing the BMC depth or the size of memory can positively affect formal runtimes.

For all configurations of RIDECORE, we observed substantial reductions in the model checking times when using updated SAT and SMT solvers. These improvements illustrate gains that come for free as SAT and SMT solvers continue to improve. These improvements do *not* require any changes to the CoSA workflow. For the default configuration and Lingeling, the improved version of Boolector results in a 7.78X speedup to find the bug (1658s vs. 213s) and a 6.35X speedup to prove correctness (4739s vs. 746s). Similarly, for the configuration with only positive clock-edge behavior and smaller data memory, swapping out Lingeling and using CaDiCaL instead in the improved version of Boolector results in speedups of 1.74X and 1.81X, respectively (150s vs. 86s and 1062s vs. 568s). With CaDiCaL, we achieved the shortest model checking times in our experiments across all configurations, SAT solvers, and versions of Boolector (bold face). Moreover, the model checking times using CoSA are comparable with those using Questa (Table I).

V. CONCLUSIONS

We presented symbolic quick error detection (SQED) as a powerful formal approach to pre- and post-silicon design verification and validation. SQED relies on bounded model checking (BMC) to check a universal property of the design. The universal property is design-independent and hence does not have to be formulated by the user. In a case study, we applied SQED to RISC-V processor cores and demonstrated that improvements to the SAT and SMT solvers that underlie the SQED workflow result in a reduction of the model checking times. We highlight that SQED addresses a main limitation of formal verification techniques: labor-intensive property

TABLE IV
APPLYING SQED TO RIDECORE USING DIFFERENT SAT SOLVERS (CADICAL, LINGELING) IN THE SMT SOLVER BOOLECTOR INSIDE COSA.

	BMC Depth	*Using CaDiCaL in Boolector (Improved)*		*Using Lingeling in Boolector (Improved)*		*Using Lingeling in Boolector (Base)*	
Design Setup/Modifications	**k**	**T (b)**	**T (c)**	**T (b)**	**T (c)**	**T (b)**	**T (c)**
Default: minimal changes to wire up QED module	23	226	697	213	746	1658	4739
Use only positive-edge clock behaviors	13	98	623	127	634	257	1771
Positive-edge clock, data memory shrinking	13	**86**	**568**	150	1062	282	1156

formulation. It thus greatly reduces the technical barrier for non-experts to apply formal techniques in practice. Moreover, our automated generator-based approach further increases usability and thereby enables rapid applications of SQED to different designs and ISAs. In light of the results of our experiments with SQED, we provide a strongly affirmative answer to our initial question whether there is hope for successful applications of formal techniques at design scale.

REFERENCES

[1] H. D. Foster, "Trends in functional verification: a 2014 industry study," in *Proc. of DAC*. ACM, 2015, pp. 48:1–48:6.

[2] M. Bohr, "The new era of scaling in an SoC world," in *IEEE ISSCC, Digest of Technical Papers*. IEEE, 2009, pp. 23–28.

[3] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[4] S. Katz, O. Grumberg, and D. Geist, ""Have I written enough properties?" - A method of comparison between specification and implementation," in *Proc. of CHARME*, ser. LNCS, L. Pierre and T. Kropf, Eds., vol. 1703. Springer, 1999, pp. 280–297.

[5] M. Emmer, Z. Khasidashvili, K. Korovin, and A. Voronkov, "Encoding industrial hardware verification problems into effectively propositional logic," in *Proc. of FMCAD*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 137–144.

[6] S. Ray and W. A. Hunt Jr., "Deductive verification of pipelined machines using first-order quantification," in *Proc. of CAV*, ser. LNCS, R. Alur and D. A. Peled, Eds., vol. 3114. Springer, 2004, pp. 31–43.

[7] A. Slobodová, J. Davis, S. Swords, and W. A. Hunt Jr., "A flexible formal verification framework for industrial scale validation," in *Proc. of MEMOCODE*, S. Singh, B. Jobstmann, M. Kishinevsky, and J. Brandt, Eds. IEEE, 2011, pp. 89–97.

[8] S. Berezin, E. M. Clarke, A. Biere, and Y. Zhu, "Verification of out-of-order processor designs using model checking and a light-weight completion function," *FMSD*, vol. 20, no. 2, pp. 159–186, 2002.

[9] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Proc. of CAV*, ser. LNCS, D. L. Dill, Ed., vol. 818. Springer, 1994, pp. 68–80.

[10] A. Reid et al., "End-to-end verification of processors with ISA-Formal," in *Proc. of CAV*, ser. LNCS, S. Chaudhuri and A. Farzan, Eds., vol. 9780. Springer, 2016, pp. 42–58.

[11] D. Lin, E. Singh, C. W. Barrett, and S. Mitra, "A structured approach to post-silicon validation and debug using symbolic quick error detection," in *Proc. of ITC*. IEEE, 2015, pp. 1–10.

[12] E. Singh, D. Lin, C. W. Barrett, and S. Mitra, "Logic bug detection and localization using symbolic quick error detection," *IEEE Trans. CAD*, pp. 1–1, 2018.

[13] RIDECORE, "Github," 2017, https://github.com/ridecore/ridecore.

[14] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. of STOC*, M. A. Harrison, R. B. Banerji, and J. D. Ullman, Eds. ACM, 1971, pp. 151–158.

[15] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. FAIA, vol. 185. IOS Press, 2009.

[16] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean satisfiability solvers and their applications in model checking," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2021–2035, 2015.

[17] M. Y. Vardi, "Boolean satisfiability: theory and engineering," *Commun. ACM*, vol. 57, no. 3, p. 5, 2014.

[18] K. Fazekas, A. Biere, and C. Scholl, "Incremental inprocessing in SAT solving," in *Proc. of SAT*, ser. LNCS, M. Janota and I. Lynce, Eds. Springer, 2019, pp. 136–154.

[19] A. Biere, "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2018," in *Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions*, ser. Department of Computer Science Series of Publications B, M. Heule, M. Järvisalo, and M. Suda, Eds., vol. B-2018-1. University of Helsinki, 2018, pp. 13–14.

[20] C. W. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking.*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 305–343.

[21] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , BtorMC and Boolector 3.0," in *Proc. of CAV*, ser. LNCS, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 587–595.

[22] C. W. Barrett et al., "CVC4," in *Proc. of CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 171–177.

[23] B. Dutertre, "Yices 2.2," in *Proc. of CAV*, ser. LNCS, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 737–744.

[24] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proc. of TACAS*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[25] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. of TACAS*, ser. LNCS, R. Cleaveland, Ed., vol. 1579. Springer, 1999, pp. 193–207.

[26] C. Mattarei et al., "CoSA: Integrated verification for agile hardware design," in *Proc. of FMCAD*, N. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, pp. 1–5.

[27] M. Heule, M. Järvisalo, and M. Suda, "SAT competitions," http://www.satcompetition.org/, 2019, accessed: 2019-07-25.

[28] 13th International Satisfiability Modulo Theories Competition, "SMT-COMP," www.smtcomp.org, 2019, accessed: 2019-07-25.

[29] M. Gario and A. Micheli, "PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms," in *Proc. of SMT Workshop*, 2015, pp. 373–384.

[30] D. Lin 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.

[31] E. Singh et al., "Symbolic QED pre-silicon verification for automotive microcontroller cores: Industrial case study," in *Proc. of DATE*. IEEE, 2019, pp. 1000–1005.

[32] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–41, 1996.

[33] ARM, "Cortex-A15," https://tinyurl.com/y75sejwz.

[34] GitHub, "Symbolic QED generator demo," 2019, https://github.com/upscale-project/generic-sqed-demo.

[35] Vscale, "Github," 2017, https://github.com/ucb-bar/vscale.

[36] GitHub, "Single-instruction checking demo," 2019, https://github.com/upscale-project/ridecore-si-checking.

[37] U. Krautz et al., "Automatic verification of floating point units," in *DAC*. ACM, 2014, pp. 151:1–151:6.

[38] RIDECORE, "Issue: bugs in rs_mul," 2017, https://tinyurl.com/y8otzyxb.

[39] U. Krautz, M. Wedler, W. Kunz, K. Weber, C. Jacobi, and M. Pflanz, "Verifying full-custom multipliers by boolean equivalence checking and an arithmetic bit level proof," in *ASP-DAC*. IEEE, 2008, pp. 398–403.

[40] A. A. R. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining Gröbner basis with logic reduction," in *Proc. of DATE*, L. Fanucci and J. Teich, Eds. IEEE, 2016, pp. 1048–1053.

[41] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *Proc. of FMCAD*, D. Stewart and G. Weissenbacher, Eds. IEEE, 2017, pp. 23–30.