AUTO-SQED:

AUTOMATED SYMBOLIC QUICK ERROR DETECTION (SQED)

FOR FORMAL VERIFICATION

By

MARIO JOHN SROUJI

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

STANFORD UNIVERSITY
Department of Computer Science

JUNE 2020

ACKNOWLEDGMENT

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.

I would like to acknowledge Florian Lonsing, Makai Mann, Yahan Yang, and Eshan Singh for their collaboration and input during my work.

AUTO-SQED:

AUTOMATED SYMBOLIC QUICK ERROR DETECTION (SQED)

FOR FORMAL VERIFICATION

Abstract

by Mario John Srouji, M.S.
Stanford University
June 2020

: Clark Barrett

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. 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. In past work, we have demonstrated how SQED can automatically find logic and security bugs in a variety of designs, and report on bugs found and efficiency gains realized in academic and industry designs. Here, I present a python based Verilog-Generator for an improved SQED module that greatly reduces the amount of manual effort and knowledge required by the designer. This tool is presented as a package of scripts, called the Automated SQED Generator (Auto-SQED). The goal is to provide a package that can: (i) almost completely separate the need for formal verification knowledge to leverage SQED, and (ii) remove the need to program directly in Verilog to create the SQED modules for a given ISA specification.

TABLE OF CONTENTS

Chapter One

Introduction

Please note that much of the introduction and background information in this paper is adapted from our ICCAD 2019 publication **"Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED"**. If you would like to learn more about the different cutting-edge formal techniques being applied, please refer to that paper [Lonsing et al., 2019].

Pre-silicon verification accounts for a significant fraction of overall design effort [Foster, 2015]. 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 [Bohr, 2009], as integrated circuits (ICs) increase design complexity tremendously to meet speed and energy targets. Examples include the use of GPUs; domain-specific accelerators; multiple processor cores; and, uncore components such as on-chip cache controllers, memory controllers, and network controllers.

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 [Clarke, Grumberg, and Peled, 1999]. 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 [Katz, Grumberg, and Geist, 1999]. 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.

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 reduced. To this end, we previously presented *symbolic quick error detection (SQED)*, as an easy-to-apply formal technique that has been successfully used for pre- and post-silicon design verification and validation [Lin et al., 2015; Singh et al., 2018]. 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.

To reduce the effort required even further, I present a vast improvement in the process of creating formal properties, far beyond the original intent of SQED alone. I present this improvement as a package of scripts, called the Automated SQED Generator (Auto-SQED). My goal was to provide a tool that can: (i) almost completely separate the need for formal verification knowledge to leverage SQED, and (ii) remove the need to program directly in Verilog to create the SQED modules for a given ISA specification. This generation technique will be discussed in detail in the remainder of the paper.

Chapter Two

Background

2.1 Overview

In the following, I will present the basic concepts behind symbolic quick error detection (SQED) as presented in [Lonsing et al., 2019]. I will start with an explanation of its predecessor, quick error detection.

2.2 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* [Lin et al., 2014]. 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)* [Lin et al., 2014]. 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 instruction, 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 the original and the corresponding duplicate registers and memory locations hold the same values. Such a system state is called *QED-consistent*. Duplicated instructions may be interleaved with original ones when the instructions of a QED test are executed. However, duplicated instructions must be executed in the same relative order as the corresponding original ones. 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.x

2.3 Symbolic Quick Error Detection

Symbolic quick error detection (SQED) [Singh et al., 2018; Lin et al., 2015] combines QED transformations with *bounded model checking (BMC)* [Biere et al., 1999]. 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.

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 the following *universal property*: if a QED test is run from a QED-consistent state, then

```
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
```

Figure 2.1 Opcode constraint example: 32-bit register-type `ADD` of RISC-V ISA.

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. 2.1 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

```
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;
```

Figure 2.2 Pseudocode for the new QED module.

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

2.4 New QED Module

In the following, I will review the New QED Module [Singh et al., 2019] that improves upon previous implementations [Singh et al., 2018]. 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. 2.2.

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 and sent through the pipeline of the processor.

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. 2.1). Being able to choose the values of that signals freely is crucial to produce QED tests where original and duplicate instructions are interleaved. 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. 2.3 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

```
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;
```

Figure 2.3 Pseudocode for the QED-ready logic of the new QED module.

design-dependent, as it has to be customized to different pipeline implementations. Function `is_write_to_original_space` is applied to input signal `write_address`, which represents the register or memory address of the data to be written. The result of the function call is stored in signal `is_original` to indicate whether the destination address of the write operation is an original or duplicate location. The counters of original and duplicate commits are incremented only when the write operation is valid (input signal `write_valid`). For simplicity, our assumption is 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. 2.3 is only applicable to single-core systems where

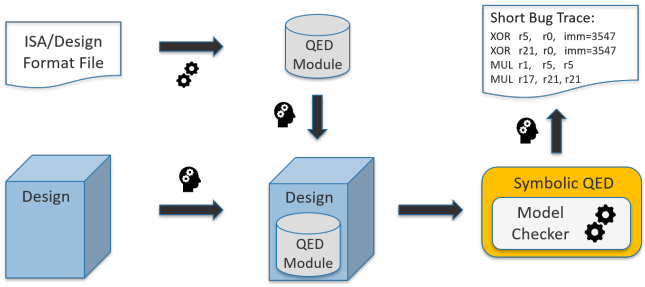

Figure 2.4 Workflow of generator-based SQED.

at most one instruction commits per cycle. The RISC-V cores we use in our experiments all have this property. For simplicity, our approach does not consider more complex designs.

2.5 Auto-SQED Generator

Although the implementation of the QED module (Fig. 2.2) must be adapted to a given DUT with respect to both the QED-ready logic (Fig. 2.3) and the opcode constraints of the instructions in the ISA (Fig. 2.1), its basic functionality, that is, duplicating instructions, is design-independent. I exploit this design-independence in a generator-based workflow implemented in Python for processor cores. The generator takes a structured description of the core and produces a QED module. Fig. 2.4 illustrates the workflow. I will use RIDECORE [RIDECORE, 2017], a RISC-V core, as a running example. A walkthrough for this example is also available in a GitHub repository [GitHub, 2019].

As a quick overview (more details in the following chapters), the format file 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. Fig. 2.5

```
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
...
```

Figure 2.5 RISC-V format file example (excerpt).

shows part of the file for a RISC-V design. 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`). I list the opcodes for all instructions in the ISA grouped by types such as register (`_R`) or immediate-type.

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, while the input format file needed by Auto-SQED is only 150 lines, without comments. The input text file is substantially easier to create than the direct Verilog implementation of SQED as well, since the majority of the information needed should be in the ISA specification sheet, and because the structure of the input format file is intuitive and flexible. The generated code by Auto-SQED compiles and synthesizes without error or need for modification, and is easily human readable with good coding style. The most crucial components of SQED, the opcode constraints (which prevent spurious counterexamples), are generated automatically (cf. generated opcode constraints for `ADD` in Fig. 2.1 and the respective opcode information in the format file in Fig. 2.5).

Given the generated QED module, we can wire it up to the design, connecting it to the instruction fetch stage. This currently requires manual effort (as with implementing the QED-ready logic, as shown in Fig. 2.3). 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.

Chapter Three

Design Considerations of Auto-SQED

3.1 Initial Design Considerations

Compiling SQED modules from a given specification is an approach that is alternative to directly implementing a generic SQED module in Verilog. Such a generic SQED module can be tailored towards a given, particular design via parameters that allow for further configuration.

The specification of the ISA is provided in terms of a structured text file that defines the different instruction formats, opcodes, and many other important aspects of an ISA. The goal is to automatically generate the Verilog source of the SQED module for a particular design based on the structured specification file of the ISA. Hooking up the generated SQED module to the given design hardly can be automated, and will have to be done manually by the user.

We identified that the QED modules for the different designs we currently have mostly differ in the instruction constraints source files, while they are otherwise quite similar to each other. Based on discussions, we had envisioned two possible ways to implement a generic QED module. The first consists of a generic implementation in Verilog that is parameterized by design specific properties such as instruction format, opcodes, instruction types, etc. The second is a compilation approach where the designer specifies the ISA of the given design in

a predefined, structured format that we formulate. We then implement a tool that parses the specification file provided by the designer in our format and automatically generate the Verilog sources of the QED module for the given design. The difference between these two approaches is that the parameterization is either reflected directly in Verilog, or in the specification file from which customized Verilog sources are generated. I chose to go the second route.

3.2 Analysis of SQED Structure

In this section I will describe how I analyzed the structure of SQED Verilog code, in order to create a general template that the Auto-SQED framework can utilize to generate SQED Verilog from an input ISA format file. I have tried to keep the description of my analysis concise for practical reasons, however I include excerpts of the Auto-SQED generated Verilog for each type of file present in SQED. This is to give the reader an idea of what the automatically-generated code looks like, and to give a better understanding of how SQED can be generalized (into a template structure).

3.2.1 Instruction Constraints File

The first file I will analyze is the instruction constraints file. This file defines the constraints necessary to determine whether a given instruction is an ISA-defined instruction type and instruction. A property is generated at the bottom of the file to ensure that these constraints are followed. An excerpt of the generated code is shown in Fig. 3.1. In the excerpt you will see example constraints generated for the immediate instruction types in RISCV RIDECORE. The overall structure of the instruction constraints file involves decoding the different bit-fields inside of the given instruction, using these bit-fields to determine which kind of instruction type and instruction it is (from the ones defined in the input ISA format file), constraining the definition of that instruction, and finally writing a property to define the allowed instructions

```
assign shamt = instruction[24:20];
assign opcode = instruction[6:0];
assign rs2 = instruction[24:20];

assign FORMAT_I = (rs1 < 16) && (rd < 16);
assign ANDI = FORMAT_I && (funct3 == 3'b111) && (opcode == 7'b0010011);
...
assign ADDI = FORMAT_I && (funct3 == 3'b000) && (opcode == 7'b0010011);
assign ALLOWED_I = ANDI || SLTIU || SRLI || SLTI || SRAI || SLLI || ORI || XORI || ADDI;

always @(posedge clk) begin
  assume property (ALLOWED_I || ALLOWED_R || ALLOWED_NOP);
end
```

Figure 3.1 An excerpt of Auto-SQED generated Verilog for the instruction constraints file.

in SQED.

3.2.2 Decoder File

The qed decoding file is used to decode the important bit-fields from a given instruction, which are then passed around to the other SQED Verilog files. Hence the only thing needed in this file is the ability to take in an instruction, and parse the different bits inside of it according to the ISA defined bit-fields. These bit-field definitions are included by the user in the input format file (described in more detail later on). An excerpt of the Auto-SQED generated qed decoder file can be seen in Fig. 3.2.

3.2.3 Instruction Modify File

The instruction modify file in SQED is in charge of taking in a given original instruction, and turning it into a SQED instruction (by changing the registers it operates on, memory, etc...). For a given instruction, we may need to both modify the registers and memory, and

```
assign shamt = ifu_qed_instruction[24:20];
assign imm12 = ifu_qed_instruction[31:20];
...
assign rs1 = ifu_qed_instruction[19:15];
assign rs2 = ifu_qed_instruction[24:20];
...
assign IS_I = (opcode == 7'b0010011);
assign IS_R = (opcode == 7'b0110011);
```

Figure 3.2 An excerpt of Auto-SQED generated Verilog for the instruction decode file.

```
...
assign NEW_rd = (rd == 5'b00000) ? rd : 1'b1, rd[3:0];
assign NEW_rs1 = (rs1 == 5'b00000) ? rs1 : 1'b1, rs1[3:0];
assign NEW_rs2 = (rs2 == 5'b00000) ? rs2 : 1'b1, rs2[3:0];
...
assign INS_R = funct7, NEW_rs2, NEW_rs1, funct3, NEW_rd, opcode;
...
assign qed_instruction = IS_I ? INS_I : (IS_R ? INS_R : qic_qimux_instruction);
```

Figure 3.3 An excerpt of Auto-SQED generated Verilog for the instruction modify file.

then be able to reassemble that instruction with the new bit-fields. For this file, we need to be able to take in the original instruction, the bit fields that were decoded earlier, and be able to output the modified SQED version. See excerpt in Fig. 3.3.

3.2.4 Instruction Mux File

In the SQED design, we require an instruction mux file in order to properly decide whether to deploy an original, or duplicate instruction through the core's pipeline. Hence this file takes in both the original and modified instruction, along with signals sent from a counter (installed inside of the core's pipeline) that keeps track of whether to deploy a duplicate instruction. It outputs the required instruction. The only thing we need to tailor in this file

```
assign qed_ifu_instruction = ena ? ((exec_dup == 1'b0) ? ifu_qed_instruction : qed_instruction) //
: ifu_qed_instruction;
```

Figure 3.4 An excerpt of Auto-SQED generated Verilog for the instruction mux file.

```
...
always @(posedge clk) begin
  if (rst) begin
    address_tail <= 7'b0;
    address_head <= 7'b0;
  end
  else if (insert_cond) begin
    i_cache[address_tail] <= ifu_qed_instruction;
    address_tail <= address_tail + 1;
  end
  else if (delete_cond) begin
    address_head <= address_head + 1;
  end
end
...
assign qic_qimux_instruction = insert_cond ? ifu_qed_instruction : (delete_cond ? instruction : 32'b1111111);
```

Figure 3.5 An excerpt of Auto-SQED generated Verilog for the qed instruction cache file.

(for a given ISA) is the size of the instruction. See Fig. 3.4

3.2.5 Instruction Cache File

The qed instruction cache file has logic to decide when to insert or remove an instruction from the cache, as well as logic to move around and reset pointers in the cache. The cache is used to store the original instructions, and later on remove them when we start deploying duplicate SQED instructions. See excerpt in Fig. 3.5.

```
  ...
  qed_decoder dec (.ifu_qed_instruction(qic_qimux_instruction),
                   .shamt(shamt),
                   .imm12(imm12),
                   .rd(rd),
                   .funct3(funct3),
                   .opcode(opcode),
                   .imm7(imm7),
                   .funct7(funct7),
                   .imm5(imm5),
                   .rs1(rs1),
                   .rs2(rs2),
                   .IS_I(IS_I),
                   .IS_R(IS_R));
  ...
  qed_i_cache qic (.qic_qimux_instruction(qic_qimux_instruction),
                   .vld_out(vld_out),
                   .clk(clk),
                   .rst(rst),
                   .exec_dup(exec_dup),
                   .IF_stall(stall_IF),
                   .ifu_qed_instruction(ifu_qed_instruction));
...
```

Figure 3.6 An excerpt of Auto-SQED generated Verilog for the top SQED file in the design, that instantiates the required modules for SQED from the aforementioned files (the instruction modify module is omitted here (...) for brevity).

3.2.6 SQED Top Level File

Finally, the top file for SQED (qed.v) is responsible for instantiating all of the aforementioned modules (in the described files), instantiating the required input/output/wire signals, and properly connecting the modules together through these signals. The Auto-SQED generator effectively generates this Verilog code for a given ISA, and outputs very well organized code as in Fig. 3.6.

Chapter Four

Design of Auto-SQED

4.1 Format File Structure and Format Parser

4.1.1 Format File

The input format file must describe all of the required signals that the Auto-SQED generators need in order to properly generate the SQED Verilog for a given ISA. The user must create this file in order to use Auto-SQED. Luckily, the way the file is structured should make it much easier for a user to create an ISA format file versus writing the direct Verilog implementation, by giving the user a structured template to follow. In addition, the information needed in the template of the format file should be readily and easily available in an ISA specification sheet. As long as a user can abide by the rules of the format file parser (described in the next section), Auto-SQED can properly generate accurate, complete, and concise Verilog. Hence this method of generalization for SQED should greatly reduce the amount of effort, and complex knowledge of both formal verification as well as Verilog required to use the SQED method. Here I will be using the RISCV RIDECORE ISA (32-bit) in order to illustrate an example of how an input format file for a given ISA can look like. I will describe the different sections, signals, and rules that are present in creating a format file through this example. Note that the format file parser supports python-style comments using the "#" character.

The format file consists of different sections that need to be defined. Each section consists

```
SECTIONS = ISA QEDCONSTRAINTS REGISTERS MEMORY BITFIELDS INSTYPES INSFIELDS INSREQS R I LW SW NOP
```

Figure 4.1 The SECTIONS line inside of the format file for RISCV RIDECORE 32-bit ISA.

```
_ISA
num_registers = 32
instruction_length = 32
pipeline_depth = 6
active_low = 1
```

Figure 4.2 The ISA section inside of the format file for RISCV RIDECORE 32-bit ISA.

of information organized according to different required ISA information. All of the sections as a whole divide up the format file into a template. This structure is also present in the internal data structure that the format parser creates (by having sections as the "key" into the first layer dictionary). We define all of our sections in the format file, in order, separated by a single space character. Sections are defined using _NAME, where NAME is the name of the section, defined later on in the given order in the format file. Note the relative ordering of these sections is not important, as long as they exist both here, and later on in the format file in the same relative order. The user must define the following sections inside of the "SECTIONS" (Fig. 4.1) keyword as the first line of the format file (first non-comment line):

1. ISA (Fig. 4.2): You need to define this section in order to include ISA-specific information like number of registers, instruction length, pipeline depth, active low, etc... You can add more definitions (or even new fields with definitions) here if you wish to extend the generators. Please continue reading the information for the next sections to understand what a "field", "definition", and "CONSTRAINT" is (as defined by the format parser) for a given format file.

2. QEDCONSTRAINTS (Fig. 4.3): This section defines what constraints to apply onto

```
_QEDCONSTRAINTS
# Divides register space in half for duplicate instructions
half_registers = 1
# Divides memory in half and uses unsigned addresses
half_memory = 1
```

Figure 4.3 The QEDCONSTRAINTS section inside of the format file for RISCV RIDECORE 32-bit ISA.

```
_REGISTERS
rd
rs1
rs2
```

Figure 4.4 The REGISTERS section inside of the format file for RISCV RIDECORE 32-bit ISA.

the SQED (i.e half of the registers are for duplicate instructions, half of the memory, etc...). you most likely do not need to change this for future ISA's.

3. REGISTERS (Fig. 4.4): Defines which bit field names in the ISA are in reference to the registers that a given instruction operates on. This section reflects changes in both the constraints, and modify file for SQED. Both the REGISTERS and MEMORY sections show the first use of "fields" inside of the format file. As you can see in Fig. 4.4, "rd", "rs1", and "rs2" are all examples of a "field" because they do not take on a specific value (using "=") like a definition would. Fields can be used to store naming information that the Auto-SQED generators can use to match up with other information prersent later on in the format file.

4. MEMORY (Fig. 4.5): Defines which bit field names in the ISA are in reference to the memory fields in a given instruction, that are to be changed in the modify file for SQED.

```
_MEMORY
imm12
imm7
```

Figure 4.5 The MEMORY section inside of the format file for RISCV RIDECORE 32-bit ISA.

```
_BITFIELDS
funct7 = 31 25
funct3 = 14 12
rd = 11 7
rs1 = 19 15
rs2 = 24 20
opcode = 6 0
shamt = 24 20
imm12 = 31 20
imm7 = 31 25
imm5 = 11 7
```

Figure 4.6 The BITFIELDS section inside of the format file for RISCV RIDECORE 32-bit ISA.

5. BITFIELDS (Fig. 4.6): Defines the different bit fields for a given instruction, and where they are located in the instruction in terms of bit range (using inclusive bounds, index 0 is the LSB of the instruction). Here we see use of "definitions" inside of the format file, because we are assigning a value to a given name using the "=" operator (as defined by the Auto-SQED parser). As an example (seen in Fig. 4.6) the "funct7" naming is assigned a value of "31 25", which will be stored in the internal data structure by the parser, and later on utilized by the Auto-SQED generators as the bit-ranges for the "funct7" field name.

6. INSTYPES (Fig. 4.7): This defines the instruction types for the ISA.

7. INSFIELDS (Fig. 4.8): This defines which bit fields are present in each instruction

```
_INSTYPES
# Putting constraints here affects
# the modify file in SQED to define the memory
# instructions. This notation involves
# classifying the ISA instruction types into types
# that the tool understands. For example, we classified
# LW and SW as MEMORYTYPE.
CONSTRAINT MEMORYTYPE,LW,SW
CONSTRAINT IMMEDIATETYPE,I
CONSTRAINT REGISTERTYPE,R
CONSTRAINT NOPTYPE,NOP

# Notice here we are defining 'fields' such as
# 'R', 'I', etc... without assigning values. See
# ../FormatParsers/format_parser.py for more info
# on how this is structured in the tool data-structure.
R
I
LW
SW
NOP
```

Figure 4.7 The INSTYPES section inside of the format file for RISCV RIDECORE 32-bit ISA. I also included the comments to show how the format parser allows interlacing of comments with fields and definitions.

type (in the same order as they exist in the instruction).

8. INSREQS (Fig. 4.9): This section defines the requirements to determine what instruction type a given instruction is, based on the bit fields. Putting constraints in this section affects the decoder file for SQED.

9. INS (Fig. 4.10): The INS section (we replace the name INS with the name of the desired ISA instruction type) specifies all ISA instructions that fall under the INS instruction type. Inside, there are fields for each instruction associated with that instruction type, with each instruction field containing definitions for the bit-field names (with values)

```
_INSFIELDS
R = funct7 rs2 rs1 funct3 rd opcode
I = imm12 rs1 funct3 rd opcode
LW = imm12 rs1 funct3 rd opcode
SW = imm7 rs2 rs1 funct3 imm5 opcode
NOP = imm12 rs1 funct3 rd opcode
```

Figure 4.8 The INSFIELDS section inside of the format file for RISCV RIDECORE 32-bit ISA.

that identify the given ISA instruction.

4.1.2 Format Parser Functionality

The ISA Format Parser inside of Auto-SQED reads in the format file I described in the previous section, and creates an internal data structure that is utilized in the remainder of Auto-SQED Verilog generation. The format parser python file has a display function that can be invoked, in order to allow the user to see a printout of how their format file has been formatted internally (and should help with debugging). The parser maintains a data structure that is inherently similar to the format of the ISA file.

The data structure is a Python dictionary with all of the "sections" defined inside of "SECTIONS" as the initial layer of keys inside the dict. Each section inside of the format file can have additional "fields" (as described previously). In this case, another layer of dictionary is created for that particular field for a given section. As an example, dict[section][field] in this case contains the data associated with the field "field" inside of section "section". "Definitions" and "CONSTRAINTS" are the final layer of the data structure. The names of the definitions and constraints are stored as keys, with their "values" being the value for the key in the dict. Hence, dict[section][field][definition] will give the value for a particular definition, associated with a given section, field, definition. See (Fig. 4.11) for an example of what the internal dictionary inside dict["INSTYPES"] looks like (here INSTYPES is the

```
_INSREQS
R
# Put the constraints here for each
# instruction type so that they are added
# into the decoder file as part of
# the IS_ins decoding process (see qed_decoder.py).
# This is an easy way to add manual constraints, we
# did not need it here.
opcode = 0110011

# Once again, 'I' here is a 'field' inside of 'section'
# 'INSREQS'. The 'field' 'I' has definitions 'opcode',
# with value '0010011'
I
opcode = 0010011

LW
opcode = 0000011
funct3 = 010

SW
opcode = 0100011
funct3 = 010
```

Figure 4.9 The INSREQS section inside of the format file for RISCV RIDECORE 32-bit ISA. Here you can see an example of using both fields to organize each instruction type, and definitions pertaining to that field (type).

section, and we access that section's information).

It is important to note that all CONSTRAINTS for a given section, or field are grouped together for that respective layer. Hence if I define four separate CONSTRAINTS as in (Fig. 4.7) for the INSTYPES section, all four of those constraints will be grouped together into a single list for that given section, and can be accessed as dict["INSTYPES"]["CONSTRAINTS"]. Even if no CONSTRAINTS are defined for a given section or field, a CONSTRAINTS key will be added into that respective layer of dictionary with an empty list of constraints.

A similar grouping method is performed when a user defines multiple values for a given

```
# Adding constraints in the instruction types
# below affects the constraints file output for SQED.
_R
# Adding a constraint right here puts it as an
# instruction or "format" type constraint for SQED.

# This is a 'field' called 'ADD', which is
# actually the name of the ADD instruction in the ISA.
ADD
# Putting a constraint here puts it
# as a constraint for the specific instruction 'ADD'
# in the constraints file for SQED. Below we defined the
# bit field values for an ADD instruction (given from the ISA spec).
funct3 = 000
funct7 = 0000000
opcode = 0110011
...
```

Figure 4.10 The R section inside of the format file for RISCV RIDECORE 32-bit ISA is an example of a section for a given instruction type.

definition. For example, let's say hypothetically that a given instruction type "I" (immediate) can take on six different opcode values to be identified. An example of this can be seen inside of the ORBIS32 ISA format file (Fig. 4.12), where I show an excerpt from the format file, and the subsequent internal representation. As seen in the example, you can define multiple different values for the same definition name, and they will all be grouped together into one list.

4.2 Auto-SQED Verilog Generators

4.2.1 Generator Overview

Up until this point, I have described the structure and template of the SQED Verilog source files, the input ISA Format File, and the Auto-SQED Format Parser. This ordering was

```
{'CONSTRAINT': ['MEMORYTYPE,LW,SW', 'IMMEDIATETYPE,I', 'REGISTERTYPE,R', 'NOPTYPE,NOP'],
'I': {'CONSTRAINT': []},
'SW': {'CONSTRAINT': []},
'LW': {'CONSTRAINT': []},
'R': {'CONSTRAINT': []},
'NOP': {'CONSTRAINT': []}}
```

Figure 4.11 Printout from the Format File Parser when accessing the upper-level dict[INSTYPES]. All of this data is consistent with what you see in the format file for the INSTYPES section.

purposeful, as Auto-SQED is designed to allow for:

1. Ultra-efficient and effective automated generation of Verilog code; The Auto-SQED toolkit can generate the complete Verilog source files required for ISA's similar to RISCV in less than a second. In addition, the format file length for RISCV RIDECORE (as an example) is only 150 lines, and is able to generate the 450 lines of Verilog code required to run SQED. Creating the format file should be at least an order of magnitude easier (in my personal opinion), than requiring the knowledge of both Verilog and Formal Verification, since a typical ISA spec-sheet should contain all information necessary. As long as the user abides by the simple rules described for the format parser, the user has the power to create complex Verilog logic for SQED. In a new case-study, I was able to add support to a new ISA (ORBIS32), which carries some similarity to RIDECORE, without any additional modifications to the parser, or generators. A new input ORBIS32 ISA format file was all that was needed to effectively create all of the SQED files needed. The tool is built with enough functionality from both a parser and generator perspective to hopefully handle most, if not all new ISA's similar to the RISCV ISA (CISC is not currently supported). This functionality was carefully designed and built based on three seperate ISA case-studies (RIDECORE, ORBIS32, BLACKPARROT), by examining the short-comings of the tool at the time.

```
Format File Excerpt:
I
# Here we see an example of an 'OR' type
# of usage for the definitions. Hence,
# opcode6 could take on any of the defined values.
# The typical 'AND' type of usage would involve
# having multiple different definitions (names)
# with possibly more than one value (which would also
# be the usage for an OR).
opcode6 = 100111
opcode6 = 101001
opcode6 = 101100
opcode6 = 101010
opcode6 = 101011
opcode6 = 101110

Internal Structure:
{'I': {'CONSTRAINT': [], 'opcode6': ['100111', '101001', '101100', '101010', '101011', '101110']}, ...}
```

Figure 4.12 Defining six different opcodes for the "I" instruction type (immediate) inside of the ORBIS32 ISA file.

2. Code error and fault isolation; The tool is designed to isolate errors between the generation of different SQED Verilog source files. The parser is designed to handle potential user-errors inside of the input format files, in a manner that "ignores" user-input versus crashing. In addition, the parser will display messages to the user if it detects some pieces of information that are incomplete, are missing, are erroneously formatted, or are in the incorrect order inside of the input ISA format file. These errors will either force the parser to end the Auto-SQED process immediately (if severe enough), or continue with the generation process if minor. The allowed-errors will later on cause subsequent generators to throw error messages to the user depending on what is going on. By allowing the generators to handle the errors on their own, the tool allows for more fine-grained error detection (since each generator knows what

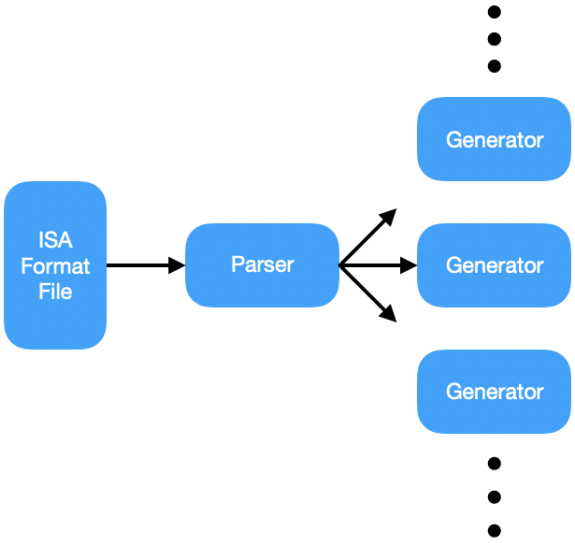

Figure 4.13 Auto-SQED is designed to be simple on a high-level, with the following architectural blocks. The ISA Format File is input into the parser, which creates the internal data structure that is passed as input to a scale-able number of SQED Verilog generators. Currently there are generators for the instruction constraints, modify, decode, and top-level file (which are all that is needed for SQED).

information it needs better than the parser would), and for partial successful generation of SQED Verilog source code (better to have 95% of your SQED Verilog automated than 0%).

3. Flexible and scale-able tool properties; I designed the toolkit to allow interested developers to be able to contribute to the code-base easily. This will ensure that the tool can continue to expand in terms of both functionality, and support of new ISA's. The goal is to make sure that the tool gives both easy-usage, and easy-extension properties for a developer.

4.2.2 SQED Generator Template

See (Fig. 4.13) for an architectural-view of the Auto-SQED toolkit. The generators for Auto-SQED ultimately are designed based off of the same principals and logic I have been describing up to this point. Just as we abstracted each SQED Verilog file into the main logic

blocks they consist of (what the files do, what kind of ISA information we need, etc.), the Verilog generators create the SQED Verilog automatically, by utilizing the ISA information stored inside of the internal data structures. This further exemplifies the importance of the Format File and Format Parser. The input ISA format file must encompass all of the ISA information needed by the generators, be structured enough to allow the parser to properly read and organize the data, and lenient enough to allow users to easily fill out the necessary sections.

Each generator (for instruction constraints, modify, decode, and the top file) operate in a fairly similar manner. Hence I think it would be more useful to describe the higher-level code design and logic here, and leave the reader to go through the actual generator Python files for more fine-grained details. Please note that chapter 3 goes hand-in-hand with this information, as it describes the structure of each SQED file (that is generated). The generators are able to leverage a small code-library created specifically for generating compile-able, synthesize-able, and human-readable code. I created this library in order to separate the process of generating logic, from the actual Verilog code implementation. This allows the generator code to be shorter, more concise, and without the clutter of Python string manipulation to generate code blocks. The interface to this Verilog string library exposes functions that can create blocks for inline conditionals, assign statements, wire initialization, bit concatenation, properties, "always" blocks, etc. Whatever is needed by the generators in terms of being able to write out Verilog to a file, should be supported by this interface. This interface code can be seen in module_interface.py under the Interface directory. For each generator, the logic is divided up into separate Verilog generation blocks:

1. Verilog module header (with inputs and outputs). Each generator creates the Verilog module header with the necessary input and output signal names, with proper formatting. The naming of these signals is leveraged from both the ISA format file, and from Auto-SQED specific naming conventions. For example, the bit-fields of an instruction are

named in the Verilog code according to the names provided in the BITFIELDS section of the format file, while signals defining a particular instruction type are preceded by the "IS_" prefix (e.g IS_R indicates a register type instruction).

2. Internal input, output, and wire initialization. Each SQED Verilog file needs to instantiate all signals (with varying bit-sizes) used in the subsequent Verilog logic blocks. This process is automated depending on the structure and input/output requirements of the given SQED module, as described in chapter 3.

3. Generator-specific Verilog logic. The SQED modules each have a template structure that is abstracted by their respective generator. The generator leverages the information located in the many sections of the input format file. For example, the instruction constraints file goes through all of the instruction type sections, and for each section reads in all individual ISA instructions stored in the fields of that section, creating the constraints based on the definitions of the bit-fields within that field. Review of the concepts in chapter 3, and the code base will reveal more generator-specific details.

4. Output signal assignments. The generators must make sure to properly assign values to all output signals of the module, particularly after the main generator-specific logic is completed.

Chapter Five

Auto-SQED Usage and Support

5.1 Code-Base

Clone from Github: https://github.com/upscale-project/sqed-generator. The only setup you need to get started with the Auto-SQED generator is Python. This tool supports the usage of both Python2.7 and Python3.x. To run the generation scripts, you need to define a format file similar to the ones in the FormatFiles directory. To run the main script, go to the Generators directory and run the generate_sqed.py script as in the following example:

python generate_sqed.py [FORMATFILE PATH] [OPTIONAL OUTPUTDIR PATH]

1. FORMATFILE PATH: Relative path to the format file for the desired ISA. Some examples are included in the FormatFiles directory.

2. OPTIONAL OUTPUTDIR PATH: Relative path to directory that will hold the resulting outputted SQED files. This directory does not need to exist, as it will be created if it does not. If you do not specify the otuput directory path, then the tool will use the default created QEDFiles directory.

I encourage you to read through the code in this directory to get a feel for how the tool's logic works. I would first start by reading the format_parser.py code in the FormatParsers

directory. This will help you understand how the tool parses and structures the relevant information from the ISA format file, which is a structured file that details the specification of a given ISA. To give you an idea of how the code-base is structured, I will describe the location and purpose of each relevant file. For the main Auto-SQED generators, see the Generators directory:

1. generate_sqed.py - This script is the 'main' script of the tool. It is in charge of parsing the command line args from the user (with error correction), calling the parser on the specified format file (and doing basic sanity checks and error detection), making sure the format file data has all the needed sections and information for the generators, launching each of the generators with the necessary inputs (and doing error detection), and finally outputting the verilog.

2. constraint_generator.py - This script is in charge of generating the instruction_constraints.v file for the SQED module, which defines the constraints for each instruction in the ISA.

3. decoder_generator.py - This script is in charge of generating the qed_decoder.v file for the SQED module, which decodes which instruction type a given instruction is in the given ISA spec.

4. modify_generator.py - This script is in charge of generating the modify_instruction.v file for the SQED module, which takes in a given ISA instruction, and modifies it's fields in order to turn it into a SQED instruction (operating on different registers or memory).

5. qed_generator.py - This script is in charge of generating the qed.v file for the SQED module, which is the 'main' file, and instantiates all of the other required SQED modules described above, as well as the instruction mux and cache verilog files (which are automatically copied over / modified by the tool inside of the generate_sqed.py script).

For the Python String-Verilog Interface, there is a script inside of the Interface directory called module_interface.py. In this script, I have all of the internal tool-functions in charge of transforming common Verilog syntax into Python strings. As an example, there are functions that allow you to add 'assign' statements, define modules, add conditionals, assign bit vectors, etc. This is where you should add Verilog-String functionality in the future. See the Generator code files for more concrete examples of how this script is leveraged.

5.2 Currently Supported ISA's

The main barrier to CPU design is that it requires expertise in several specialties: electronic digital logic, compilers, and operating systems. Because of this, commercial vendors such as ARM Holdings and MIPS Technologies charge royalties for the use of their designs, as well as patents and copyrights. They also often require non-disclosure agreements before releasing documents that describe their designs' detailed advantages (like ISA-spec sheets). RISC-V was started to solve these problems, with the goal to make a practical ISA that was open-sourced, and usable academically in any hardware or software design without royalties required. A need for a large base of contributors is part of the reason why RISC-V was engineered to fit so many use-cases. The designers assert that new principles are becoming rare in instruction set design, as the most successful designs of the last forty years have become increasingly similar. Hence, a well-designed open instruction set designed using well-established principles should attract long-term support by many vendors. The RISC-V Instruction Set Architecture (ISA) sheets are also released ands available publicly, hence allowing the use of SQED for Formal Verification. That is why our main ISA that we focus on for SQED and Auto-SQED is on RISC-V Architectures (or variants).

Currently, we have direct support for the following ISA's (and are confident that any ISA variants of RISC-V like these can easily be supported/extended with Auto-SQED as well):

1. RISC-V RV32 RIDECORE ISA.

2. ORBIS-32 ISA.

3. BLACKPARROT ISA (32-bit).

Extension to 64-bit architectures is currently supported, as the user can simply change the instruction size argument inside of the input ISA Format File, and accordingly define the bit-field sizes later on.

REFERENCES

Biere, Armin et al. (1999). "Symbolic Model Checking without BDDs". In: *Proc. of TACAS*. Ed. by Rance Cleaveland. Vol. 1579. LNCS. Springer, pp. 193–207. ISBN: 3-540-65703-7.

Bohr, Mark (2009). "The new era of scaling in an SoC world". In: *IEEE ISSCC, Digest of Technical Papers*. IEEE, pp. 23–28. ISBN: 978-1-4244-3458-9. DOI: 10.1109/ISSCC.2009.4977293.

Clarke Jr., Edmund M., Orna Grumberg, and Doron A. Peled (1999). *Model Checking*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-03270-8.

Foster, Harry D. (2015). "Trends in functional verification: a 2014 industry study". In: *Proc. of DAC*. ACM, 48:1–48:6. ISBN: 978-1-4503-3520-1. DOI: 10.1145/2744769.2744921.

GitHub (2019). *Symbolic QED generator demo*. https://github.com/upscale-project/generic-sqed-demo.

Katz, Sagi, Orna Grumberg, and Daniel Geist (1999). ""Have I written enough Properties?" - A Method of Comparison between Specification and Implementation". In: *Proc. of CHARME*. Ed. by Laurence Pierre and Thomas Kropf. Vol. 1703. LNCS. Springer, pp. 280–297. ISBN: 3-540-66559-5. DOI: 10.1007/3-540-48153-2_21.

Lin et al., David (2014). "Effective Post-Silicon Validation of System-on-Chips Using Quick Error Detection". In: *IEEE Trans. CAD* 33.10, pp. 1573–1590. DOI: 10.1109/TCAD.2014.2334301.

Lin, David et al. (2015). "A structured approach to post-silicon validation and debug using symbolic quick error detection". In: *Proc. of ITC*. IEEE, pp. 1–10. ISBN: 978-1-4673-6578-9. DOI: 10.1109/TEST.2015.7342397.

Lonsing, F. et al. (2019). "Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED: Invited Paper". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8.

RIDECORE (2017). *GitHub*. https://github.com/ridecore/ridecore.

Singh et al., Eshan (2019). "Symbolic QED Pre-silicon Verification for Automotive Microcontroller Cores: Industrial Case Study". In: *Proc. of DATE*. IEEE, pp. 1000–1005. ISBN: 978-3-9819263-2-3. DOI: 10.23919/DATE.2019.8715271.

Singh, E. et al. (2018). "Logic Bug Detection and Localization Using Symbolic Quick Error Detection". In: *IEEE Trans. CAD*, pp. 1–1. ISSN: 0278-0070. DOI: 10.1109/TCAD.2018.2834401.