Finding and Understanding Bugs in FPGA Synthesis Tools

Yann Herklotz
yann.herklotz15@imperial.ac.uk
Imperial College London
London, UK

John Wickerson
j.wickerson@imperial.ac.uk
Imperial College London
London, UK

ABSTRACT

All software ultimately relies on hardware functioning correctly. Hardware correctness is becoming increasingly important due to the growing use of custom accelerators using FPGAs to speed up applications on servers. Furthermore, the increasing complexity of hardware also leads to ever more reliance on automation, meaning that the correctness of synthesis tools is vital for the reliability of the hardware.

This paper aims to improve the quality of FPGA synthesis tools by introducing a method to test them automatically using randomly generated, correct Verilog, and checking that the synthesised netlist is always equivalent to the original design. The main contributions of this work are twofold: firstly a method for generating random behavioural Verilog free of undefined values, and secondly a Verilog test case reducer used to locate the cause of the bug that was found. These are implemented in a tool called Verismith. This paper also provides a qualitative and quantitative analysis of the bugs found in Yosys, Vivado, XST and Quartus Prime. Every synthesis tool except Quartus Prime was found to introduce discrepancies between the netlist and the design. In addition to that, Vivado and a development version of Yosys were found to crash when given valid input. Using Verismith, eleven bugs were reported to tool vendors, of which six have already been fixed.

CCS CONCEPTS

• **Hardware → Electronic design automation**; **Hardware description languages and compilation**; • **Software and its engineering → Software verification and validation**.

KEYWORDS

fuzzing, logic synthesis, Verilog, test case reduction

ACM Reference Format:
Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375310

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375310

```
1 module top (y, clk, w1);
2   output y;
3   input clk;
4   input signed [1:0] w1;
5   reg r1 = 1'b0;
6   assign y = r1;
7   always @(posedge clk)
8     if ({-1'b1 == w1}) r1 <= 1'b1;
9 endmodule
```

Figure 1: Vivado bug found automatically by Verismith. Vivado incorrectly expands `-1'b1` to `-2'b11` instead of `-2'b01`. The bug was reported and confirmed by Xilinx.[1]

1 INTRODUCTION

Almost all digital computation performed in the world today relies, in one way or another, on a logic synthesis tool. Computation specified in RTL passes through a logic synthesis tool before being implemented on an FPGA or an ASIC. Even designs that are expressed in higher-level languages eventually get synthesised down to RTL. Computation that is executed in software is carried out on a processor whose design has also, at some point, passed through a logic synthesis tool.

These tools are not only *pervasive*: they are *trusted*. That is, any bugs they contain undermine efforts to ensure the correctness of hardware designs. For instance, the Silver processor has been formally proven to implement its ISA correctly [15], and the Kami platform enables hardware designs to be formally verified using Coq [5]. Yet in both cases, the final hardware is only as reliable as the logic synthesis tool that produces it. That these tools are trusted is explicitly acknowledged – Silver's correctness proof assumes that the "toolchain taking Verilog to FPGA bitstreams is bug-free", while Kami's guarantees hold only "if we trust [the] compiler to preserve the semantics." We ask in this paper whether this trust is well placed.

Logic synthesis tools are prone to bugs because of the complexity involved in performing the aggressive optimisations that are required to meet power consumption and timing demands. Moreover, these bugs are likely to be particularly egregious because they can be hard to detect. This is especially the case when synthesising large designs, because post-synthesis simulation or verification is often skipped, or is only performed towards the end of the development cycle, due to time constraints. Even when these bugs *are* detected during post-synthesis testing, the root cause can be extremely challenging to isolate and work around [16]. With hardware designs growing ever larger and increasingly being created by software

[1] https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Unsigned-bit-extension-in-if-statement/td-p/981789

Tool	Vendor	License	Versions
XST [25]	Xilinx	Commercial	14.7
Vivado [24]	Xilinx	Commercial	2016.1, 2016.2, 2017.4, 2018.2, 2019.1
Quartus Prime [12]	Intel	Commercial	19.2
Quartus Prime Lite [12]	Intel	Commercial	19.1
Yosys [23]	Yosys	ISC License	0.8, 0.9, 3333e00, 70d0f38

Table 1: Versions of the synthesis tools that were tested.

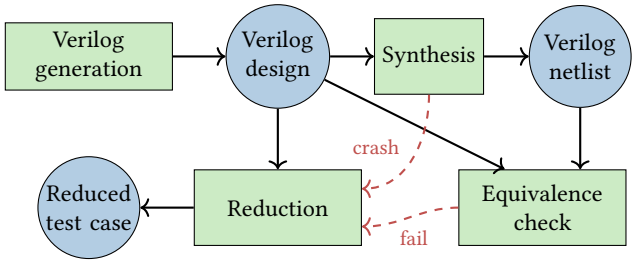


Figure 2: Overview of the testing approach used in Verismith by generating random Verilog. If the synthesis tool crashes or the equivalence check fails, the test case is reduced into a minimal representation, shown by the red dashed arrows.

engineers operating HLS tools, this inability to debug is becoming ever more troublesome.

In this paper, we describe the design and implementation of a tool called Verismith [9] for finding and understanding bugs in logic synthesis tools that target FPGAs. Verismith generates pseudo-random, valid, deterministic Verilog designs, feeds each to a synthesis tool, and uses an SMT solver or the ABC [3] circuit verification tool to check that the output is logically equivalent to the input. If they are not equivalent, it has found a bug. Verismith then iteratively reduces the Verilog design with the aim of finding the smallest (and hence most understandable) program that still triggers a bug. An example of such a bug that was found by Verismith is shown in Figure 1. The test case has been tweaked manually for readability, but it was found and reduced automatically by Verismith.

We ran Verismith on five major synthesis tools for FPGAs for a total of 18000 CPU hours, as can be seen in Table 1. We found two classes of bugs: Verilog designs that cause the synthesis tool to produce incorrect output, and Verilog designs that cause the synthesis tool to crash. We reported a total of 7 unique test cases that are mis-synthesised: 4 in Yosys, 3 in Vivado. We also reported 3 unique crash bugs: 1 in Yosys and 2 in Vivado. In addition, a bug was also found in Icarus Verilog [21], which is a simulator used in Verismith to check counterexamples returned by the equivalence check. All 11 test cases have been reported to the manufacturers; the Yosys and Icarus Verilog bugs have all been confirmed and fixed, whereas the Vivado bugs have been confirmed and are awaiting a fix. Testing Quartus Prime proved difficult, however, once it worked, we did not find any failing test cases, meaning it was quite stable. Failing test cases found in XST were not reported, as it is no longer being actively maintained.

The three main contributions of this paper are the following:

- We present an algorithm for generating random, valid, deterministic Verilog designs that employ a variety of combinational and behavioural constructs, shown in Section 3.
- We explain how to check whether a given Verilog design triggers a bug in a synthesis tool under test in Section 4 and then present an algorithm for reducing Verilog designs to find the smallest program that triggers the bug in Section 5.
- Finally, in Section 6, we report the results of synthesising our generated programs using three logic synthesis tools and evaluate how our design decisions affect the ability of Verismith to find bugs in these tools.

Verismith is fully open source and can be found on GitHub.[2]

2 OVERVIEW OF VERISMITH

Verismith is the implementation of the Verilog generation and test case reduction algorithm with the goal of finding bugs in synthesis tools. Figure 2 shows the main workflow in Verismith. First, a random design is generated and passed to the synthesis tool, which should produce an equivalent netlist. If the synthesis tool crashes, a bug has been found and the initial design would therefore be reduced to a minimal test case that still triggers the crash. This is shown by the red dashed arrow which shortcuts an error occurring in synthesis to the reduction step. However, if synthesis completes successfully, the netlist is compared to the initial design. If they differ, the resultant design needs to be reduced as well, which is depicted by the other red dashed arrow.

Verismith generates semantically correct and deterministic Verilog, meaning that it should always pass synthesis and the values of output wires should be uniquely determined by values of the input wires. An equivalence check can therefore be performed between the generated design and the synthesised netlist to determine if the synthesis was correct. If the netlist is shown not to be equivalent to the design, it must mean that there is a synthesis bug, as false negatives and false positives are not possible. However, there is the possibility that the SMT solver does not give an answer in time, in which case it cannot be determined if the design is equivalent to the netlist.

Verismith was implemented in Haskell because its algebraic data types are well-suited for capturing the syntax of a language like Verilog, and its pure functions make it easier to reason about and test functions.

3 GENERATING VERILOG

To test the synthesis tools, valid random Verilog needs to be generated so that the synthesis tool successfully produces a netlist that can be compared to the original design.

3.1 Target language

The synthesisable subset of Verilog 2005 [10] was chosen as the target HDL as it is widely supported. Every generated Verilog file

[2] https://github.com/ymherklotz/verismith

contains a list of module definitions with an arbitrary number of inputs and parameters, and one output. The module body consists of a list of module items, which can be any of the following constructs:

- continuous assignment,
- local parameter declaration,
- module instantiation,
- always and initial block, and
- wire and variable declaration.

Inside always blocks and initial blocks, behavioural Verilog can then be generated, which supports the following constructs:

- blocking and nonblocking assignments,
- conditional statements, and
- for loops.

Finally, many different expressions are supported, such as:

- all synthesisable unary and binary operators,
- bit selection,
- function calls,
- concatenation, and
- ternary conditional operators.

The most notable features that are missing from this grammar subset are function and task definitions, but we expect adding them to be straightforward. In addition to that, the synthesisable subset of Verilog specifies many constructs that should be ignored by synthesis tools, so these are also not generated by Verismith. These are constructs that do not have a direct hardware equivalent and include delays, specify blocks or system task calls. Although initial blocks and variable initialisation are supposed to be ignored by the synthesis tool, except when modelling ROMs [10, section 7.7.9.1], both of these features are supported in all the synthesis tools that were tested for FPGAs, as these can be powered up into a known state. These are therefore used by Verismith to set the design in a known state at the start.

3.2 Properties of generated programs

The data types specifying the syntax strictly follow the grammar, with the result that only syntactically valid programs can be represented. However, this cannot guarantee that the programs are semantically valid as well.

We define semantically valid programs as ones that should be accepted by simulators and synthesis tools without producing any errors or critical warnings. Even though random programs following the grammar will pass the parsing stage of these tools, they will most likely error out when they are processed. To ensure semantic validity, several rules have to be followed, including:

- module inputs have to be nets,
- no continuous assignments to variables or blocking/nonblocking assignments to nets occur, and
- every variable that is used has to be declared.

Moreover, the generated Verilog should be deterministic. This simplifies the equivalence checking stage, as there cannot be any false positives or false negatives. Therefore the equivalence check indicates if there is a bug or not. In addition, we argue that bugs found using purely deterministic Verilog are more severe than bugs that include undefined values, as most Verilog in production is deterministic and it is generally bad practice to write Verilog that depends on undefined values. Furthermore, undefined values might mask bugs in the synthesis tool by allowing it to optimise away large sections of the design. Therefore, other Verilog patterns have to be avoided even though they are semantically and syntactically correct. For example, some constructs that lead to nondeterministic behaviour are the following:

- driving a wire from two different sources,
- dividing by zero,
- passing fewer bits to a module than it expects,
- selecting bits that are out of range, and
- using a net that has not been declared previously.

```
[probability]
  expr.binary = 5
  expr.concatenation = 3
  expr.number = 1
  expr.rangeselect = 5
  expr.signed = 5
  expr.string = 0
  expr.ternary = 5
  expr.unary = 5
  expr.unsigned = 5
  expr.variable = 5
  moditem.assign = 5
  moditem.combinational = 1
  moditem.instantiation = 1
  moditem.sequential = 1
  statement.blocking = 0
  statement.conditional = 1
  statement.forloop = 1
  statement.nonblocking = 3

[property]
  module.depth = 2
  module.max = 5
  output.combine = false
  sample.method = "random"
  sample.size = 10
  size = 20
  statement.depth = 3
```

Figure 3: Configuration file used to tweak properties of the generated program including frequencies of constructs.

3.3 Generation algorithm

To generate Verilog that avoids these constructs, the syntax tree is built sequentially, line by line, using a context to keep track of important facts about previously generated code. This inherently prohibits combinational loops, as all the values being assigned to the current wire will already have been assigned beforehand. The context contains: (1) a list of nets and variables that are declared and assigned in the current context, (2) a list of modules that can safely be instantiated and (3) a list of parameters that are available in the module. These are used to safely create module items and make sure that they do not introduce undefined values or race conditions. When creating a new variable assignment, the nets can be safely taken from the context as these are guaranteed to have been assigned previously.

The context also contains relative frequencies which are attached to each construct that can be generated. These determine how often a construct will appear in the output. We tweaked these manually using a configuration file, shown in Figure 3, to obtain a good coverage of all the features, while keeping the synthesis and equivalence checking time to a minimum. In particular, maximum statement list length and depth were heavily tweaked to reduce the nesting depth of statements, as that would increase the synthesis time and equivalence checking time exponentially. Operations like divide and modulo were removed for most of the testing, because with nets or variables containing a large number of bits, the circuits generated by the synthesis tools were too large to be efficiently optimised and checked for equivalence.

The output of the Verilog generation is a Verilog file containing multiple modules. The entry point is a top-level module. For every module, a random number of inputs with random sizes are chosen and added to the context. A clock for sequential blocks and an output port are also added. Random parameters are also declared and added to the context so that they are available in any expressions inside the module. Finally a list of module items are generated.

To ensure that expressions remain deterministic, extra checks are performed to ascertain that no undefined values are added to an expression. Checks can either be performed statically at generation or dynamically at runtime. For example, if bits are selected from a net, the size of the net is checked against the range of the selection statically. However, runtime checks need to be added to operations like division to check against division by zero. This is similar to the safe math wrappers that are used in an existing C fuzzer called Csmith [26] to avoid undefined behaviour like signed overflow.

Once the module items have finished generating, all the internally declared variables and nets are concatenated and assigned to the output, so that any discrepancies in the internals of the module are detected by the formal verification step. An example of a generated module is shown in Figure 4, which declares many variables and concatenates them to the output y. Figure 4 also shows the different sections that are created by Verismith. First come all the declarations of the nets and variables that are used and assigned to somewhere in the body of the module. Then follows the assignment of the internal state of the module to the output wire y, so that any errors in any of the assignments will be detected in the output. Finally, the main body of the module contains a list of random constructs, which were generated according to the configuration file that was passed to Verismith.

Remark. If the number of IO ports is limited, because the design first needs to fit onto an actual FPGA such as in Quartus Prime Lite, the output can be reduced to one bit by applying the XOR reduction operator. For example, the concatenation

```
assign y = {reg3,reg4,reg5,wire6,wire7,reg8,wire9};
```

can be changed to

```
assign y = ^{reg3,reg4,reg5,wire6,wire7,reg8,wire9};
```

This continues to hold all the necessary information to detect a single bitflip in the internal state of the module, however, synthesis

```
module top #(parameter param0 = 5'h9e23848124)
(y, clk, wire0, wire1, wire2, wire3);
  // *** Declarations ***
  output wire [(5'h31):(1'h0)] y;
  input wire [(1'h0):(1'h0)] clk;
  input wire [(3'h6):(1'h0)] wire0;
  input wire [(4'ha):(1'h0)] wire1;
  input wire signed [(4'ha):(1'h0)] wire2;
  input wire [(4'hb):(1'h0)] wire3;
  reg [(3'h2):(1'h0)] reg20 = (1'h0);
  reg [(3'h5):(1'h0)] reg19 = (1'h0);
  reg [(3'h4):(1'h0)] reg18 = (1'h0);
  reg [(2'h2):(1'h0)] reg17 = (1'h0);
  reg [(4'ha):(1'h0)] reg16 = (1'h0);
  reg signed [(4'h9):(1'h0)] reg15 = (1'h0);
  wire [(3'h6):(1'h0)] wire5;
  wire [(2'h3):(1'h0)] wire4;
  // *** Assign output ***
  assign y =
    {reg20,reg19,reg18,reg17,reg16,reg15,wire5,wire4};
  // *** Random module items ***
  assign wire4 = (((~wire1) ? ((((15'h9ecc51592fdeb04)
    ? reg17[(5'h2):(2'h2)] : (reg18 ? wire2 : wire0))
    ? $unsigned(((-2'ha73a956341f45c0) << reg18)) :
    wire1[(4'ha):(3'h7)]) - reg18) :
    reg15[(4'h9):(3'h7)]) >>> $unsigned($signed((
    reg16[(4'ha):(3'h7)] ? ((wire1 && reg16) &&
    {reg15, reg15, wire3}) : (reg18 ? (~&wire3) :
    (-39'ha7a1419cd4ea34a))))));
  assign wire5 = $signed(((wire2 ? (
    (-8'h5e411249da4f335) ? (4'hb2fa97daeae9ff) :
    wire1) : (wire4 ? wire2 : wire1)) ?
    $signed(wire3) : ({(7'hbac46141008d14)} >>>
    (&wire0))));
  always @(posedge clk) begin
    for (reg15 = (1'h0); (reg15 < (2'h2)); reg15 =
        (reg15 + (1'h1))) begin
      if (((wire3 == (~(reg16 + wire1))) >=
          {$signed(wire0[(2'h2):(1'h0)])}))
        reg16 <=  ($unsigned($unsigned(wire1)) <
          wire3[(1'h1):(1'h1)]);
      else reg16 <= $unsigned(reg17[(2'h2):(2'h0)]);
      reg17 <=  wire3[(1'h0):(1'h0)];
    end
    reg18 <= $signed(({wire0} ~^ wire3));
  end
  always @(posedge clk) begin
    if (wire3[(4'h9):(3'h6)])
      reg19 = $signed($unsigned(wire1)) <<
        $unsigned({wire1});
    reg20 <=  ({{{(~|wire3), $unsigned(reg19)} ?
      reg16 : reg15[(2'h2):(1'h1)]),
      (~&((wire0 ? wire3 : reg17) ~^ reg18))}
      || ((~&(wire3[(4'hb):(4'h9)] ? wire4 : (+wire5)))));
  end
endmodule
```

Figure 4: Example module generated by Verismith showing the distinct sections that are produced.

and verification are much slower because of the larger circuit. We demonstrate this empirically in Section 6.

4 EQUIVALENCE CHECKING

The equivalence check is a crucial step in verifying that the synthesis tool behaved properly by proving that the synthesised netlist is equivalent to the original design. The equivalence check itself is performed using Yosys [23] and the ABC [3] back end. However, an SMT solver such as Z3 [7] can also be used as a back end to perform the equivalence check. As only deterministic Verilog is being tested, which does not contain any undefined values, the equivalence check proves that the design is equivalent to the netlist over all possible inputs.

The equivalence is checked using the following property: the output wires of the randomly generated Verilog design and the synthesised netlist should always be equal at the clock edge given the same inputs. This is expressed in Verilog by instantiating both modules and asserting that the outputs are equal:

```
module equiv( input clk, input [6:0] w0, input [10:0] w1
            , input signed [10:0] w2, input [11:0] w3 );
  wire [49:0] y1, y2;
  top t1(y1, clk, w0, w1, w2, w3);
  top_synth_netlist t2(y2, clk, w0, w1, w2, w3);
  always @(posedge clk) assert(y1 == y2);
endmodule
```

where y1 and y2 are the outputs of the design and the netlist respectively. The Verilog is passed through Yosys synthesis to obtain either SMT-LIBv2 [2] for an SMT solver, or a netlist for ABC.

As this process is performed by Yosys itself, when testing Yosys synthesis there might be bugs that are not found by the equivalence check, as the same bug will be present when design is passed to the SMT solver or to ABC. However, bugs can still be found in optimisations that Yosys only applies when it is properly synthesising the design instead of passing it to an external solver. In addition to that, if Yosys is tested with multiple other synthesis tools, the synthesised netlist produced by Yosys can be compared to the other synthesised netlist instead of the original design. Therefore, the bug in Yosys should not trigger anymore as it is only synthesising two netlists. Finally, a test bench can also be created for the generated design and the netlist by passing random test vectors to both top-level modules and checking that the output remains the same.

After the equivalence check is performed, the checker returns a counterexample which can be added to a testbench and hence simulated, to make sure that it does indeed expose a difference between the design and the netlist.

5 TEST CASE REDUCTION

Reducing an HDL is different from reducing programming languages. The time it takes to discover the presence of an error is much higher with synthesis and equivalence checking than with compilation and execution. Existing reduction methods such as delta debugging [28] or hierarchical delta debugging [18] are effective at reducing failing test cases for programming languages, but rely on a quick feedback loop which tells the reducer if the current version of the test case still triggers a bug, i.e. that it is still interesting.

Therefore, we developed a general reduction approach similar to hierarchical delta debugging to speed up the reduction of an arbitrary Verilog design. Then, due to the structure of the random test cases, an optimisation can be added to improve the efficiency when dealing with those test cases. As the reducer is tightly coupled with the generator, the original AST can be used to analyse the source and reduce it further. However, the reducer still works in a standalone manner for the supported subset of Verilog.

The main goal of the reduction algorithm is to quickly reduce the size of the program, so that it can be analysed as soon as possible and testing can be resumed. As synthesis and verification are time consuming, even for small designs, it is crucial to minimise the number of steps needed by the reduction algorithm. To achieve this, a depth-first binary search is performed on the syntax tree, with different levels of granularity. At every node, the current program is checked against the synthesis tool to check that the bug is still present.

The steps of the general reduction are detailed below. Each of the steps will result in a binary choice which is explored in a greedy fashion, meaning that the first option is taken until the bug is not present anymore. The second option is only explored if the first option would eliminate the bug from the design. If the bug is not present there either, the reduction algorithm backtracks to the last version that still contains the failing test case and terminates. Each of the following steps is repeated until it cannot be applied anymore without eliminating the bug.

(1) Half the modules, excluding the top-level module, are removed from the current Verilog file. All instantiations of modules that were removed and that are present in the left-over modules are eliminated as well. Any wires that were connected to the output ports in the module instantiations are also pruned and any references to those modules in any expressions are also removed so that no extra undefined values are added.
(2) Half of the module items are removed. For all the assignments that are removed, the declaration of the variable or net that was being assigned is also removed together with any uses of the variable or net in any expressions in that module.
(3) The same is done for all the statements inside the always blocks. Half of the assignments in each always block are removed. If a conditional statement is encountered, it is reduced by picking one of the branches, and choosing the other if the bug is not present anymore.
(4) Finally, expressions are simplified and reduced to narrow down on the bug. Concatenations are split into two, and binary operators are removed and replaced by their LHS and RHS.

By reducing the Verilog at different levels of granularity, as much code as possible is removed at every step. As a binary choice is made at every step, the reduction will converge to a result, which may, however, not be optimal.

An optimisation can be carried out if the test case was generated using Verismith, because all the internal variables and nets are concatenated and added to the only output. The binary search can then

Tool	Total test cases	Failing test cases	Distinct failing test cases	Bug reports
Yosys 0.8	26400	7164 (27.1%)	≥ 1	0
Yosys 3333e00	51000	7224 (14.2%)	≥ 4	3
Yosys 70d0f38 (crash)	11	1 (9.09%)	≥ 1	1
Yosys 0.9	26400	611 (2.31%)	≥ 1	1
Vivado 18.2	47992	1134 (2.36%)	≥ 5	3
Vivado 18.2 (crash)	47992	566 (1.18%)	5	2
XST 14.7	47992	539 (1.12%)	≥ 2	0
Quartus Prime 19.2	80300	0 (0%)	0	0
Quartus Prime Lite 19.1	43	17 (39.5%)	1	0
Quartus Prime Lite 19.1 (No `$signed`)	137	0 (0%)	0	0
Icarus Verilog 10.3	26400	616 (2.33%)	≥ 1	1

Table 2: Summary of failing test cases found in each tool that was tested.

be performed only on the concatenation, while deleting all references to variables or nets that are removed from the concatenation. Once the wire in question is found, the standard reduction algorithm can attempt to reduce it further. However, normally only one assignment is left which can consequently be identified manually.

This optimisation only helps if the test case is processed sensibly by the synthesis tool, where the netlist is produced but is not equivalent to the input. It may fail to reduce the test cases properly when the synthesis tool crashes, as it only operates on the output of the module, which may not be the cause of the crash.

Remark. Another optimisation that could be performed to increase the speed of the reduction is to use the counterexample provided by the equivalence checker in iterations of the reducer using simulation, instead of rerunning the equivalence check at every iteration.

6 EVALUATION

Verismith was run for 18000 CPU hours, testing the following synthesis tools: Vivado [24], XST [25], Quartus Prime [12], Quartus Prime Lite [12], and Yosys [23]. Our experiments were designed to answer five questions. First, how many bugs can we detect in various synthesis tools? Second, how does increasing the size of the generated Verilog designs affect the efficiency of our testing approach? Third, what is the effect of XOR-ing all the outputs of a design into a single wire? Fourth, how does the stability of synthesis tools change with each new release? Finally, how does our custom Verilog test case reducer fare against an existing delta-debugging tool called C-Reduce [20]?

Remark. Many problems were encountered when testing Quartus Prime. The lack of documentation about the d-flip-flop used by Quartus Prime required a lot of trial and error to get it working properly. In addition to that, optimisations in Quartus Prime, such as multiply-accumulate, had to be disabled as these would use hardware of the specific FPGA that was targeted, therefore using encrypted modules. Finally, to be able to generate a Verilog netlist using Quartus Prime Lite, the design had to be first fitted on a real FPGA, which meant that virtual pins had to be used to fit all the IO ports, or the output had to be combined into one bit.

6.1 How many bugs were found?

Table 2 presents the breakdown of all the bugs found in the synthesis tools. These tools were tested separately and were therefore given different test cases. The only tools where we observed any crashes were Vivado and Yosys. Crashes in Vivado are therefore shown in a separate row. The crash found in Yosys was found when briefly testing a different development version of Yosys (commit hash 70d0f38). All the other rows of the table refer only to test cases that mis-synthesise. The only tool where all test cases succeeded was Quartus Prime, even after ramping up the number of test cases to over 80,000. In a surprising contrast, we found the highest percentage of failures in its sister tool, Quartus Prime Lite. We only ran a small number of test cases many were failing. Upon inspection, we found that these failures could all be traced back to the `$signed` function, which converts an expression to signed and sign extends it if necessary. When we disabled this construction in our random generator, we found no further failures. Alongside the various Verilog synthesis tools, we also tested a Verilog simulator called Icarus Verilog, which was used to check the counter examples returned by the equivalence checker. This revealed one bug in the simulator, which was reported and fixed.

A test case failure was identified as being unique if one minimal test case could not be reduced into a different minimal test case. This often required manual intervention to go from the automatically reduced test case to the minimal test case that could be compared to the bugs that had already been identified. One caveat regarding identifying unique mis-synthesis bugs has to be noted, because there may be one bug in the synthesis tool that is identified multiple times with different unique minimal test cases. For the open source synthesis tools, this can be verified by following the bug report, however, for commercial synthesis tools, this cannot be checked. Unique crashes were easier to identify, as the tools dump a stack trace that shows the exact position where the crash occurred. Therefore, crashes at different positions could be identified as being unique.

Only the failing test cases in Yosys can be analysed properly, as the other failing test cases have either not been fixed, or the fix itself has not been disclosed. Yosys was tested using three different versions, the stable version of Yosys 0.8 before any of our bug fixes

were integrated, the master branch of Yosys (commit hash 3333e00) as it was being developed, just before our bug fixes were introduced and finally the most recent stable release which is Yosys 0.9. The master branch of Yosys was tested to assist the current development of Yosys and be able to report bugs before they affect users.

All the bugs mentioned were reported and confirmed by the tool maintainers and vendors. Most of the Yosys bugs were fixed within the day, whereas Xilinx confirmed the bug and intend to "fix this issue in future releases".[3] The following subsections show examples of bugs that were found.

6.1.1 Yosys peephole optimisation.[4] An example of a bug in Yosys is in the peephole optimisation pass as shown below. A peephole optimisation is the replacement of a specific sequence of instructions with a more efficient but equivalent sequence of instructions.

```
module top (y, w);
   output y;
   input [2:0] w;
   assign y = 1'b1 >> (w * (3'b110));
endmodule
```

The piece of code above was identified for a peephole optimisation which optimised the multiply and shift operations where one of the operands in the multiply is constant. However, the code above was optimised in a way that did not truncate the result of the multiplication, meaning that if the input is `w = 3'b100`, `y` would be set to 0 because the shift amount would be set to `6'b011000` instead of the correct value `3'b000`. Therefore, the correct output of the module should be 1 when `w` is set to `3'b100`.

6.1.2 Yosys peephole crash.[5] A crash was also found in the peephole optimisation, which is shown in the code below.

```
module top(y, clk, wire1);
   output [1:0] y;
   input clk;
   input [1:0] wire1;
   reg [1:0] reg1 = 0, reg2 = 0;
   assign y = reg2;
   always @(posedge clk) reg1 <= wire1 == 1;
   always @(posedge clk) reg2 <= 1 >> reg1[1:1];
endmodule
```

It will shift 1 to the right by one bit if the second bit in `reg1` is set. However, the code performing the optimisation did not check the vector's length before attempting to access the last element in the vector. This therefore led to the code crashing because it tried to index an element outside of the vector.

6.1.3 Vivado bug.[6] Another bug was found which ignored the bit selection. The code sample for that bug is slightly more complex.

```
module top (y, clk, w0);
   output [1:0] y;
   input clk;
   input [1:0] w0;
   reg [2:0] r1 = 3'b0;
   reg [1:0] r0 = 2'b0;
   assign y = r1;
   always @(posedge clk) begin
      r0 <= 1'b1;
      if (r0) r1 <= r0 ? w0[0:0] : 1'b0;
      else r1 <= 3'b1;
   end
endmodule
```

For an input of `w0 = 2b'10` for two clock cycles, the final output should be `2'd0`, because the if statement is entered on the second clock cycle and the LSB of `w0` is assigned to `r1`, which is `1'b0`. However, with Vivado the output is `2'10` instead, meaning Vivado does not truncate the value of the input to the LSB in `w0[0:0]`.

6.2 How does program size affect efficiency?

The efficiency of generating different sizes of Verilog code was also analysed to identify the optimal size that finds the most failing test cases. This was measured by conducting eight separate experiments, each with Verismith configured to generate programs of a different size, and each experiment given 48 hours to find as many bugs as possible.

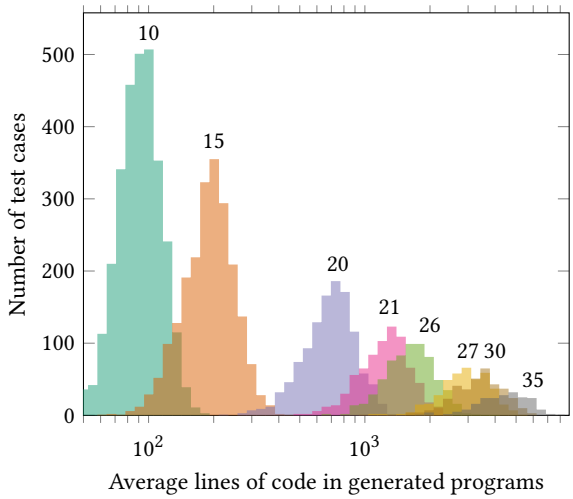
Figure 5a shows how the distribution of test case sizes is affected by the `size` parameter given to Verismith. Each of the eight distributions in the figure is labelled with the value of the `size` parameter that generated it. As the `size` parameter becomes larger, we observe that it becomes harder to control the test case size (hence the more spread out distributions) – this is because of the inherent randomness in the generation algorithm.

Figure 5b compares the eight experiments by how many test case failures they found. The average test case size in each experiment is displayed along the x-axis. We see that shorter programs are more effective at triggering mis-synthesis bugs, and longer programs are more effective at triggering crash bugs. Therefore, generating programs that have a size of around 700 lines of code might be optimal to find the largest variety of bugs.

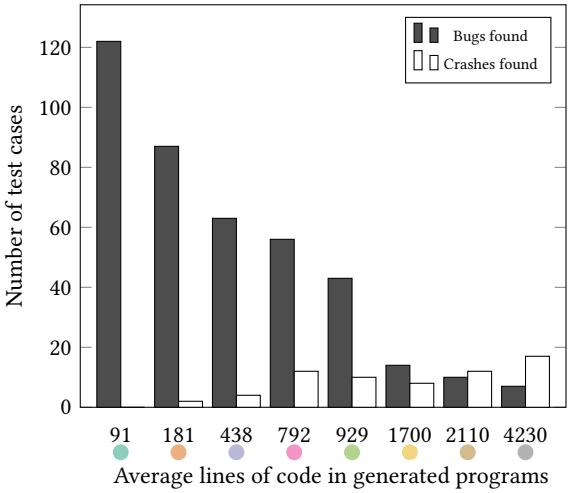
Larger inputs should find more failing test cases, as more combinations are tried that might trigger a bug in the synthesis tool. The designs also become much more complex as the size of the Verilog increases. However, it is more efficient to run multiple small random Verilog modules instead of large ones, because in the same amount of time, many more failings test cases are found. The run time of the synthesis and the equivalence checking is the limiting factor, as both of these increase exponentially with the size of the input. On the other hand, the number of crashes still increases with the size of the input. This is because a crash normally occurs at the start of the synthesis process, which means that the complexity of the input does not affect the time taken to discover the crash. Larger inputs will therefore have a greater chance of crashing the synthesis tool.

Thus, it seems that it is more useful to generate small Verilog modules which will synthesise and pass equivalence checking as

[3] https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Bit-selection-synthesis-mismatch/m-p/982632#M31484
[4] https://github.com/YosysHQ/yosys/issues/1047
[5] https://github.com/YosysHQ/yosys/issues/993
[6] https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Bit-selection-synthesis-mismatch/m-p/982632#M31484



(a) Distribution of test case sizes in each experiment.



(b) Efficiency of various sizes at finding failing test cases.

Figure 5: How the bug-finding efficiency varies with generated program size.

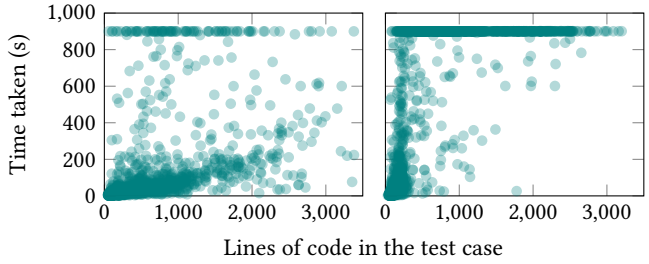


Figure 6: Synthesis and equivalence checking time as program size increases for test cases. Left: output combined using concatenation. Right: output combined to one bit using unary XOR operator.

fast as possible. However, solely generating programs of the smallest size might only result in finding the same bug repeatedly. Instead, it is better to generate Verilog designs at around 700 lines of code so that a larger variety of inputs can be created.

6.3 What is the effect of XOR-ing the outputs?

Figure 6 compares concatenating the output to combining the output into one bit, as was mentioned in Section 3.3, by comparing the time taken to perform the equivalence check. Synthesis time is not displayed, because it did not change when the output was combined into one bit or not, and scaled exponentially with size in both cases. Equivalence checking is the limiting factor when performing the random testing, because it scales exponentially but can also take much longer depending on the test case. The horizontal line of points at the very top are the test cases that timed out at 900 seconds. The graph on the left shows that equivalence checking time mostly scales exponentially with the size of the test case, with around 5% of test cases failing. However, when the output is combined into one bit, which is shown by the graph on the right, the equivalence checking time scales much worse as the lines of code of the test cases increase, with around 29% of test cases failing overall. The time taken increases in a nearly vertical line, meaning even small test cases timeout most of the time. The reason for this is that the circuit generated by the unary XOR operator takes a long time to verify, as all possible paths are explored.

Reducing the output to one bit increases the time to perform the equivalence check dramatically, making it impossible to generate programs of more than 500 lines and check for bugs that were introduced. Therefore, the output should not be combined unless it is absolutely necessary.

6.4 How stable are synthesis tools?

By fuzzing different versions of synthesis tools, the general stability of the tools can be observed. One might expect newer versions of the tool either to have fewer bugs, as they are reported and fixed, or to have more bugs, as new features are added. In total, 837 test cases were run through the four different versions of Vivado, however, 134 test cases timed out and were therefore disregarded. Figure 7 shows how many test cases produced failures in each version of Vivado. Each horizontal ribbon represents a group of test cases that produced failures in the same tools and the larger the ribbon, the more test cases followed the same trajectory.

Figure 7 shows that Vivado 2016.1 and 2016.2 have exactly the same test case failures. From 2016.2 to 2017.4, the bugs originating from one group of test cases were fixed, but two different groups of test cases starts to fail. The total number of failing test cases is also higher in 2017.4 than in previous versions. Finally, two groups of test cases are fixed for version 2018.2, but another group of test cases start to fail, which is the largest proportion of test cases in the diagram. There is one group that stays constant in all the versions of Vivado comprising 15 test cases, which are likely due to failures that have not been found or reported yet.

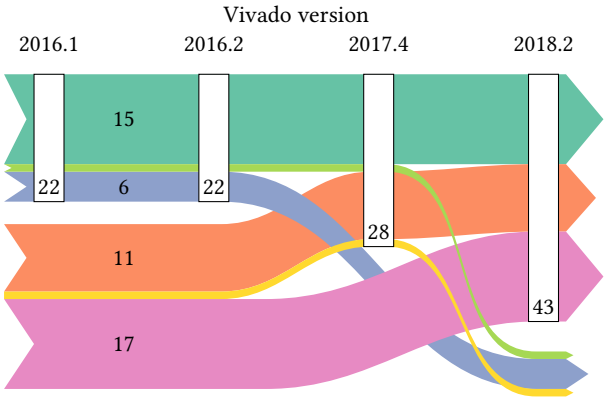


Figure 7: Tracking the same set of test cases across four versions of Vivado. The white rectangles indicate the total number of failing test cases per version. Test cases that fail in the same versions are grouped into a ribbon, and each ribbon is labelled with the number of test cases it contains. The interleaving of ribbons shows how bugs may have been introduced or fixed in each version.

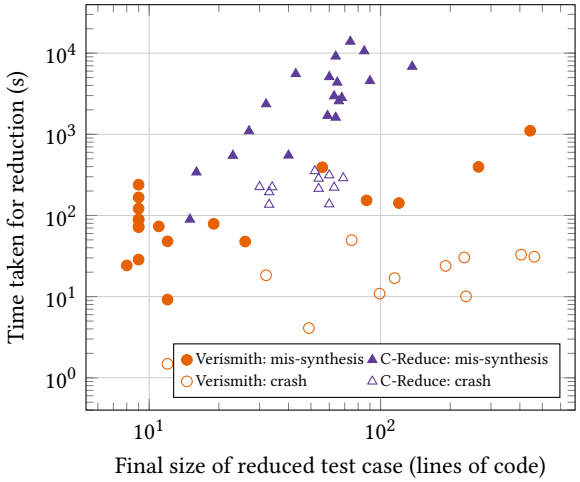


Figure 8: Comparing the effectiveness of test case reduction by Verismith and by C-Reduce using 30 randomly selected test cases that fail in Yosys. Only 24 test cases were successfully reduced by C-Reduce, whereas Verismith reduced all 30. Both axes use a logarithmic scale. Points towards the bottom left are favoured.

6.5 How effective is our reducer?

To test the efficiency of the test case reducer that was implemented in Verismith, it was compared against an existing test case reducer for C/C++, called C-Reduce [20]. C-Reduce is a general reducer containing various passes that perform different reduction algorithms. As it is tailored to reduce C-like languages, C-Reduce is much less effective at reducing unknown languages, because it cannot analyse and perform reductions on the AST. However, it can still apply other passes which are independent of the input language.

C-Reduce has a notion of test case validity and checks for undefined and unspecified behaviour when reducing C so that it can be avoided. This is necessary, as the reducer might otherwise discard the original bug and reduce the undefined behaviour instead. To be able to compare C-Reduce to Verismith, it requires a similar notion of how to avoid undefined behaviour when reducing Verilog. We added a few context-free heuristics to the script that check if the test case is still *interesting*, i.e. that the bug is still present. The heuristics are the following:

(1) Check that it can be parsed by Verismith, to guarantee that the test case is still in the supported subset of Verilog and is syntactically valid.
(2) Apply context-free heuristics, such as requiring all registers to be initialised.
(3) Check that the synthesis tool does not output any warning to catch any other errors that may have been introduced.

C-Reduce was tested against our implementation of the reduction algorithm using 30 randomly selected test cases that fail in Yosys. C-Reduce was run with a default configuration and all the C/C++ passes turned off. As per the default, C-Reduce was run in parallel on four cores. Verismith, being a single-threaded design, ran on only one core. Figure 8 shows the results of the comparison, differentiating between crashes and mis-synthesis bugs. Six out of the 30 test cases were reduced to contain only undefined behaviour using C-Reduce and discarded the original cause of the discrepancy between the netlist and the design. This was often because inputs to module instantiations were removed, which led to undefined values in those inputs. To mitigate this, context-sensitive heuristics would be needed, to make sure the right amount of inputs is present in a module instantiation. In addition to that, because C-Reduce does not know the Verilog syntax, most of the minimal test cases were unreadable and were therefore passed through Verismith to compare the final sizes properly. Figure 8 allows us to draw the following conclusions:

Verismith is much faster than C-Reduce. The average time taken by Verismith is 119s, while the average time taken by C-Reduce is 2640s (note that the logarithmic scale on the y-axis de-emphasises this discrepancy). This is expected, as Verismith performs a strict binary choice at every pass, and does not consider additional alternatives, so that the number of synthesis runs is minimised. Additionally, Verismith has access to the original AST and performs semantically valid reductions that will not introduce undefined behaviour.

Verismith reduces mis-synthesis bugs further than C-Reduce. The median size of the reduced test case for Verismith is 11 lines, whereas for C-Reduce this is 61 lines. This can be explained by the fact that Verismith has access to the AST and can therefore perform more semantically valid transformations. However, the average number of lines reduced is about the same for both tools, as there are cases where Verismith does not find the optimal reduction.

C-Reduce reduces crash bugs further than Verismith. Even though Verismith is faster than C-Reduce in most cases, C-Reduce seems to be better at reducing crashes, taking only slightly longer than Verismith but achieving a smaller reduced test case in general.

This is because Verismith always tries to perform semantically valid transformations, which is not relevant when reducing crashes. Checking that the Verilog is valid is not important as long as the tool still crashes with the original error message.

7 RELATED WORK

Random Verilog generation. VlogHammer [22] is also a Verilog fuzzer that targets the major commercial synthesis tools, as well as several simulators. It has found around 75 bugs to date which have been reported to the tool vendors. In contrast to our tool, VLogHammer does not generate programs with multiple modules and it does not support behavioural Verilog (e.g. always blocks). Whereas our tool only generates deterministic Verilog (which we have argued is the most important part of the language), VLogHammer generates nondeterministic Verilog, and requires an additional simulation step to avoid false positives. Finally, VLogHammer does not perform test case reduction, instead only generating small modules that can be analysed manually if they fail.

Another random Verilog generator is VERGEN [19], which generates behavioural Verilog by randomly combining high-level logic blocks such as state machines, MUXes and shift registers. However, because it generates these predefined constructs, it produces well-behaved code which is unlikely to test many different combinations of Verilog constructs.

American Fuzzy Lop (AFL) [27] is a general-purpose fuzzer for binaries and uses instrumentation to guide the mutation of existing test cases. However, given that synthesis tools are highly complex programs with a large number of different states, it can be difficult to identify the correct set of inputs to enable the fuzzer to find a bug. In addition to that, the fuzzer has no notion of correct behaviour, and can therefore only detect crashes. We ran AFL on Yosys for 144 CPU hours and did not find a crash.

Random generation and differential testing has also successfully been applied to fuzz various OpenCL implementations [14], including the Intel FPGA SDK for OpenCL [11].

Equivalence checking. Differential testing [17] is the standard method for checking compiler correctness, by passing the input to two or more different compilers for the same language and checking if the output behaves in the same way. If one achieves a different result, then it is assumed that there must be a bug. Csmith [26] uses this technique to check if the output of different C compilers is correct. However, as the output of synthesis tools is Verilog, it can be checked formally for equivalence with the initial design to ensure that no bugs were present in the synthesis tool.

Modern commercial logical equivalence checkers, such as Conformal [4] could also solve the problem of comparing netlists to the original design, as they are often built with that use case in mind.

Test case reduction. Test case reduction is usually performed by a process called delta-debugging [28] which splits the source code into parts using simple lexer rules and tries to remove as many parts as possible. C-Reduce [20] is an example of an advanced reduction algorithm for C-like languages that makes use of parallel executions of different subsets of the test case to identify one that is still interesting, which is reduced further.

Verified synthesis. There has also been work on verified synthesis, such as PBS [1], written in ML and verified mechanically using Nuprl [6], or Π-ware [8], which is a high level HDL in Agda where the synthesis process to gates is formally proven. Such systems should, in theory, withstand our random testing. However, it is an enormous effort to build a fully verified synthesis tool that offers comparable performance to state-of-the-art tools. Moreover, there could still be bugs in the non-verified parts of these tools – as was found to be the case with the CompCert [13] verified C compiler when tested using Csmith [26].

8 CONCLUSION AND FURTHER WORK

This paper introduced a method for behavioural Verilog generation and Verilog reduction, which was implemented in an open-source tool called Verismith. This tool successfully found and reported a total of eleven bugs in Yosys, Vivado and Icarus Verilog, which are now either fixed or confirmed and scheduled to be fixed in upcoming releases.

The main limitation of Verismith is that it cannot produce a design that contains undefined values, meaning no bugs can be found that are caused by them. Another limitation is that implementations for all the FPGA primitives are needed for the different tools that are used. Finally, only synthesis tools that can output Verilog are supported, which may limit which synthesis tools can be tested.

Further work could be done on supporting a larger subset of Verilog, which improves the testing of the synthesis tools. In addition to that, undefined values could also be supported, which would further increase the variety of test cases that could be generated. Undefined values could be introduced in a controlled manner to support this, as Verismith already has a notion of determinism. This would allow for control over how much of the output should be undefined, which would reduce the risk of undefined values masking possible bugs and affecting a large proportion of the output.

It is worth asking whether the bugs we have found using Verismith really matter, or whether they would only be triggered by code patterns that are unlikely to appear in production code. This is hard to answer definitively, but it is worth noting that one member of the Xilinx user community remarked that the bug we found in Vivado "looks to me to be a rather critical bug".[7] In general it does seem like these tools cannot be completely trusted, because they can generate a netlist that is not equivalent to the original or even crash given correct and deterministic Verilog.

It is our hope that tools like Verismith can not only be valuable to designers of logic synthesis tools as a way to catch more bugs, but can also provide designers with a safety net that gives them the confidence to implement ever more ambitious optimisations.

ACKNOWLEDGMENTS

We acknowledge financial support of a PhD studentship from the Research Institute on Verified Trustworthy Software Systems (VeTSS), which is funded by the National Cyber Security Centre (NCSC), and the EPSRC IRIS programme grant (EP/R006865/1). We thank Alastair Donaldson, Eric Eide, Martin Ferianc, Brent Nelson, and the FPGA '20 reviewers for valuable suggestions.

[7] https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Bit-selection-synthesis-mismatch/m-p/982632#M31484

REFERENCES

[1] M. Aagaard and M. Leeser. 1991. A formally verified system for logic synthesis. In *[1991 Proceedings] IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 346–350. https://doi.org/10.1109/ICCD.1991.139915

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6.* Technical Report. Department of Computer Science, The University of Iowa.

[3] Robert Brayton and Alan Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer, Berlin, Heidelberg, 24–40.

[4] Cadence. 2019. Conformal Equivalence Checker. https://bit.ly/2mkp0aa

[5] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

[6] Robert L. Constable et al. 1986. Implementing Mathematics with The Nuprl Proof Development System.

[7] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Heidelberg, 337–340.

[8] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2018. Pi-Ware: Hardware Description and Verification in Agda. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Tarmo Uustalu (Ed.), Vol. 69. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9:1–9:27. https://doi.org/10.4230/LIPIcs.TYPES.2015.9

[9] Yann Herklotz and John Wickerson. 2019. *Verismith: FPGA '20 Artifact.* https://doi.org/10.5281/zenodo.3559802

[10] IEEE Std 1364.1 2005. *IEEE Standard for Verilog Register Transfer Level Synthesis.* Standard. 1–116 pages. https://doi.org/10.1109/IEEESTD.2005.339572

[11] Intel. 2019. Intel FPGA SDK for OpenCL software technology. https://intel.ly/2WXoTj8

[12] Intel. 2019. Intel Quartus. https://intel.ly/2m7wbCs

[13] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 42–54. https://doi.org/10.1145/1111037.1111042

[14] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

[15] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. ACM, New York, NY, USA, 1041–1053. https://doi.org/10.1145/3314221.3314622

[16] William McDonald and Janny Liao. 2006. Logic Equivalence Checking has Arrived for FPGA Developers. In *Design and Verification Conference (DVCon)*.

[17] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[18] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 142–151. https://doi.org/10.1145/1134285.1134307

[19] Boris Ratchev, Mike Hutton, Gregg Baeckler, and Babette van Antwerpen. 2003. Verifying the Correctness of FPGA Logic Synthesis Algorithms. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays (FPGA '03)*. ACM, New York, NY, USA, 127–135. https://doi.org/10.1145/611817.611837

[20] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[21] Stephen Williams. 2019. Icarus Verilog. http://iverilog.icarus.com/

[22] Clifford Wolf. 2019. VlogHammer. https://bit.ly/2kCxjO3

[23] Clifford Wolf. 2019. Yosys Open SYnthesis Suite. https://bit.ly/2kAXg0q

[24] Xilinx. 2019. Vivado Design Suite. https://bit.ly/2wZAmld

[25] Xilinx. 2019. XST Synthesis Overview. https://bit.ly/2lGtkjL

[26] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[27] Michal Zalewski. 2015. American fuzzy lop. http://lcamtuf.coredump.cx/afl/

[28] A. Zeller and R. Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (Feb 2002), 183–200. https://doi.org/10.1109/32.988498