Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2019

Project Title:	**Fuzzing Verilog**
Student:	**Yann Herklotz Grave**
CID:	**01062783**
Course:	**EIE4**
Project Supervisor:	**Dr John Wickerson**
Second Marker:	**Dr David Thomas**

Abstract

All software eventually relies on hardware to function correctly. Hardware correctness is becoming increasingly important due to the growing use of custom accelerators and field-programmable gate arrays (FPGA) to speed up applications on servers. Ensuring the quality of synthesis tools is vital for the reliability of the hardware and, as a consequence, reliable synthesiser testing is essential.

This project aims to improve the quality of synthesisers by testing them using randomly generated and correct Verilog, comparing the equivalence of the design with the synthesised logic. The main contribution of this project is a method of generating behavioural and procedural Verilog that does not contain undefined behaviour, which is implemented in a tool called VeriFuzz. In addition, this project proposes a Verilog test-case reducer to locate the bugs that were found. Finally, the bugs that were found in different synthesisers are analysed using qualitative and quantitative results. Every synthesiser that was tested was found to introduce discrepancies between the logic and the design, and some synthesisers crashed when given valid input. VeriFuzz found 21 unique bugs, of which 8 were reported to tool vendors and 3 were fixed.

Acknowledgements

First, I would to thank my project supervisor Dr John Wickerson for all the advice and guidance he has given me throughout the project and for always taking the time to discuss and provide solutions to various problems I faced.

I would also thank my family for supporting me throughout my studies and always being there when I needed them the most.

Last but not least, I would like to thank my friends that have always motivated me for the past four years and have made the time at Imperial College London memorable.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AIG	and-inverter graph
ASIC	application-specific integrated circuit
AST	abstract syntax tree
DAG	directed acyclic graph
DTM	deterministic Turing machine
DUT	device under test
EBNF	extended Backus-Naur form
EMI	equivalence modulo inputs
FPGA	field-programmable gate array
HDL	hardware description language
HLS	high-level synthesis
LUT	look-up table
MCVE	minimal, complete and verifiable example
NDTM	non-deterministic Turing machine
PUT	program under test
r.e.	recursively enumerable
RAM	random access memory
ROM	read-only memory
RTL	register-transfer level

Chapter 1

Introduction

All software eventually relies on the hardware being correct and even fully verified software can experience unexpected behaviour if the hardware it is running on is faulty. Therefore, hardware bugs have a more severe impact than software bugs, as hardware cannot be patched directly, but has to rely on software patches which can affect performance. Meltdown [1] and Spectre [2] are two recent examples of hardware faults that affect billions of devices from phones to cloud services. Spectre especially affects Arm [3], Intel [4] and AMD [5] processors, and Spectre attacks break many security assumptions that other software security measures rely on. Consequently, large numbers of devices are at risk and can be attacked independent of their operating system and their software. To guard against this, software fixes have to be implemented by all the major operating systems such as Linux [6], Microsoft Windows [7], and iOS and macOS [8]. Although the software patches manage to successfully guard against these risks, there is a high price to pay for this protection in terms of performance loss. For multi-node jobs in high performance computing (HPC) clusters the performance decreases by as much as 5–11% [9]. In contrast to hardware bugs, software bugs usually affect fewer devices and even if bugs are found in firmware or in the operating system, these can usually be fixed with a software update that is unlikely to affect performance.

Another interesting illustration of this problem is the expensive recall of all the Intel Pentium processors in 1994 due to a bug in the floating point division operation FDIV [10], which "cost Intel 475 million dollars against earnings pre-tax." A software bug would have been considerably cheaper to remedy and would not have required a brand damaging recall action. A simple software update would have sufficed. This clearly shows that is of major importance for hardware to be reliable and function correctly. As a consequence efficient methods of designing hardware and the ability to test and verify correctness become a key factor for success for manufacturers.

Furthermore, as processing power increases, hardware is rapidly becoming more and more complex and the increased complexity of digital hardware such as CPUs and GPUs today necessarily demands the use of a hardware description language (HDL) [11] such as

Verilog or VHDL. It would be impossible to design these complex circuits by manually drawing the schematics or laying out the transistors. In addition to that, designs written using an HDL can be tested and synthesised to real hardware based on the same description. An alternative to writing in an HDL is to describe the hardware using high-level languages that eventually compile to an HDL using a process called high-level synthesis (HLS), which enables the description of hardware at an even higher level. For example, there are many different Risc-V [12] implementations like Rocket Chip [13] or BOOM [14], which are implemented using a hardware generation language called Chisel [15], which outputs Verilog for emulation and synthesis. Furthermore, working with an HDL allows for thorough testing of the design, and even for the use of formal verification to prove that a design complies with the specification using tools like JasperGold [16].

The use of HLS specifically has increased rapidly due to the growing demand of custom accelerators in cloud services using an field-programmable gate array (FPGA) or an application-specific integrated circuit (ASIC) to accelerate deep convolutional neural networks [17], database management [18] or digital signal processing [19]. Amazon Web Services (AWS) and Microsoft Azure each provide a way to use an FPGA to accelerate applications in the form of the F1 instance [20] and Catapult Project [21] respectively. The FPGA can be programmed using an HDL such as Verilog or VHDL, or using HLS from a language like C++. This allows for software to be accelerated by transforming it into hardware. As the hardware is specialised to the application, the performance will be higher than running the application on a CPU.

HLS is the process of transforming higher-level code to an HDL, which then gets converted to logic that implements the same behaviour as the design. To place the design onto an FPGA, the HDL design has to be translated to a netlist using a process called hardware synthesis. The netlist defines the connections between logical components available on the targeted FPGA or ASIC [22], which have to be placed on the FPGA through a process called “Place & Route”.

The synthesis step is crucial, as it transforms the higher-level description of the hardware, written in Verilog or VHDL, into logic that will be placed on the FPGA. However, if the synthesiser transforms the code into a netlist that does not have the same behaviour as the initial Verilog design, there will be serious flaws in the hardware and these will in turn affect anything that uses the hardware. In addition to that it may be nearly impossible to detect whether the generated netlist for the design is faulty or not, as the designers have to depend on the tool doing the right job. Even if the design was simulated and tested thoroughly beforehand, there is no guarantee that the synthesised netlist will have the same behaviour as the initial design. The only solution to this would be to formally verify that the netlist is equivalent to the design, however, with the large Verilog designs that are currently being synthesised to a complex CPU that might be infeasible.

It is therefore essential to test that synthesisers produce correct hardware from the HDL design. The aim of this project is to investigate the effectiveness of random testing, also called fuzzing, to test the Verilog implementation in synthesisers. However, the fuzzer should also support simulators and be able to check their correct execution of the hardware. A key component will be to devise a way to generate random, but valid, Verilog files and subsequently run these systematically through Verilog synthesisers, to examine if they are all handled correctly. If a mismatch between the synthesised netlist and the original design is found, the test-case is analysed and reduced so that it can be reported to the tool vendors. The bug will hopefully be fixed in future releases of the synthesis tools, thereby improving the quality of synthesisers.

1.1 Project Objectives

The main objective of this project is to automatically test synthesisers and improve the reliability of the hardware that they generate, as well as test simulators. To achieve this, a deterministic random Verilog generation method is introduced and implemented in a fuzzer called VeriFuzz [23]. The main goals of the fuzzer are to automate as much of the process as possible, from the random Verilog generation, to reduction of the failing test-cases that found bugs in a tool. The following conditions are necessary to achieve this objective:

- Method of generating deterministic and semantically correct random Verilog, so that the output of the synthesisers and execution of simulators can be validated.
- Support of the main synthesisers and simulators, with the ability to easily add more.
- Reduction of the Verilog code that produced an error to a minimal test-case that replicates that error, which facilitates identifying and reporting unique bugs.
- Configurable generation allowing for many different properties about the random generation to be changed. This allows for testing using different configurations and comparing them against each other.
- Generation of a report to get a quick overview of what random Verilog tests passed and which ones failed.

In addition to the implementation, another objective of this project is to provide various qualitative and quantitative results from fuzzing the different synthesisers. Unique bugs are reported to the synthesiser vendors so that these can be fixed in future versions. Overall, VeriFuzz found 21 unique bugs, out of which 8 bugs were reported to the tool vendors. Three of those bugs, which were found in a synthesiser called Yosys [24], have already been fixed, two of which were added to the Yosys regression tests and one was deemed interesting enough to be added to the main test bench.

1.2 Report Overview

The report structure is the following:

Chapter 2: Background summarizes the nature, purpose and functioning of fuzzers focusing on techniques used to fuzz compilers. Details on Verilog syntax and semantics as well as deterministic Verilog will also be given. Finally, an existing Verilog fuzzer named VlogHammer [25] will be discussed.

Chapter 3: Project Specification describes the specifications of VeriFuzz in detail and how these fulfil the project objectives.

Chapter 4: Implementation explains the design decisions made and gives details on the implementation of VeriFuzz as well as challenges that were encountered.

Chapter 5: Results contains qualitative and quantitative results obtained from fuzzing and from experiments that were conducted with VeriFuzz.

Chapter 6: Evaluation evaluates the implementation of VeriFuzz and compares it to the objectives set out at the start of this project.

Chapter 8: User Guide provides documentation on the VeriFuzz command line tool that is used for fuzzing, and also on the file structure of the VeriFuzz library.

Chapter 2

Background

This chapter surveys the necessary background information that is used throughout this report. Section 2.1 introduces Verilog, its syntax and semantics, as well as important properties that have to be considered when randomly generating hardware using Verilog. Section 2.2 surveys popular fuzzing techniques, focusing on compilers as these will also be most relevant when fuzzing simulators and synthesisers. The final section will discuss VlogHammer, an existing Verilog fuzzer for synthesisers and simulators, and how it compares to the design of VeriFuzz.

2.1 Verilog

Verilog is an HDL, which is commonly used to design and verify digital circuits. This section will focus on Verilog 2005 [26], the Verilog standard that is supported by most the simulators and synthesisers, and is also the standard that VeriFuzz targets. Verilog contains a subset that can be synthesized to hardware [27], so that, for example, it can be placed on an FPGA. The remaining Verilog language can be simulated and provides many tools that allow for formal verification. This means that a design written in Verilog can be thoroughly tested beforehand by simulating it with a test bench or formally proving properties about the code. The same design can later be synthesised to hardware with confidence that the resultant logic will behave in the same way as the tested design.

This section will cover the grammar specification of Verilog, as well as go over the different subsets of Verilog, such as syntactically correct, semantically correct, synthesisable, deterministic and chosen subset of Verilog. The chosen subset is the Verilog subset that is supported by VeriFuzz. Next, simulators and synthesisers are described and the differences between them are noted. Finally, formal verification process is explained as it is used to perform equivalence checking between the synthesised netlist and the original design.

2.1.1 Grammar specification

To make an effective tool that can fuzz the Verilog grammar, it has to be modelled inside the tool. It is therefore useful to analyse if the grammar is context-free, context-sensitive or recursively enumerable (r.e.) as it can influence the way in which the grammar is represented and what kind of guarantees result from that representation. An r.e. grammar is a language L for which there exists a deterministic Turing machine (DTM) which accepts the language and does not terminate otherwise, meaning it will accept any input $x \in L$ and will not terminate if $x \notin L$. Context-sensitive grammars, on the other hand, are a subset of r.e. and defined to be all the languages L that are accepted by a linearly bounded non-deterministic Turing machine (NDTM). Finally, context-free languages are such that their production rules can only contain one non-terminal symbol. The reason why context-free languages are particularly interesting, is because an abstract syntax tree (AST) can be defined so that it solely represents languages that are valid according to this grammar. Any instantiation of the AST will therefore produce a valid program according to the extended Backus-Naur form (EBNF) specification.

Syntactically correct Verilog is the language that is generated by the EBNF specification that is included in the standard [26]. This specification of Verilog is context-free because the EBNF definition only has a single non-terminal symbol on the left hand side. However, the fact that a piece of Verilog is syntactically correct does not mean that the simulator and synthesiser will accept it, because the EBNF grammar does not include all the other rules that are defined in prose in the standard. There are therefore quite a few subsets of syntactically correct Verilog that give stronger guarantees. Semantically correct Verilog is the subset of Verilog that follows all of the rules in the standard and should therefore be accepted by any Verilog simulator conforming to the standard. However, not all Verilog that can be simulated is accepted by the synthesiser, because many Verilog constructs such as printing to the console, do not make sense during synthesis. For this reason, a synthesisable subset of Verilog is defined [27]. Finally, the standard allows for non-determinism and undefined behaviour, which hinders the direct comparison of the synthesisers' outputs. Hence, a smaller subset can be defined that only contains the Verilog language that produces a deterministic output with synthesis and simulation. The inclusion relation between the different subsets is shown in Figure 2.1.

Context-sensitive semantically correct Verilog

Semantically correct Verilog is the subset of syntactically correct Verilog that is accepted by a simulator. An example of the difference between syntactically correct and semantically correct is shown in Listing 2.1. Both modules are syntactically correct and will be parsed correctly, however, only `mod1` is semantically correct, whereas `mod2` will not compile with a simulator. This is because the standard says that only wires can be declared implicitly, whereas variables have to be declared explicitly. As `a` is used in an always block, it has to

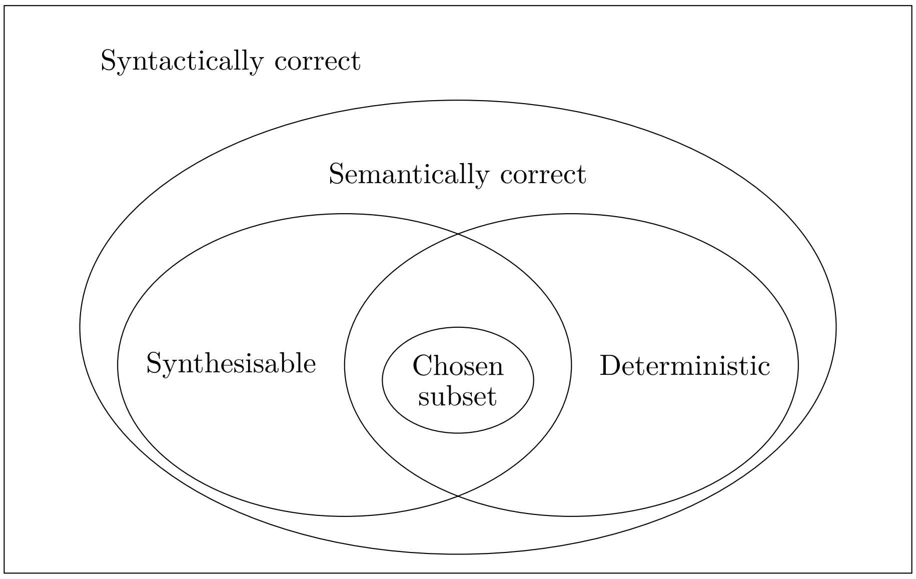


Figure 2.1: Hierarchy of Verilog subsets

be declared as a variable using the `reg` type. Therefore, even though both modules are syntactically valid, only the first module is semantically valid. This shows that a necessary condition for semantically valid Verilog is that variables have to be declared using `reg` if they are used. Instantiating a module that is not present is also not semantically valid, but would be grammatically correct.

```verilog
module mod1;
  assign a = 2;
endmodule

module mod2;
  always a <= 2;
endmodule
```

Listing 2.1: Example of context sensitivity for semantically valid Verilog programs.

Many examples of undefined behaviour are given in the standard. Undefined behaviour is also present in many other languages such as C, however, in Verilog there is first party support for undefined values using `x` or `X`. For example, the literal `3'b10x` has an undefined value as the least significant bit, which means that after synthesis it could be a 1 or a 0, whereas the simulator would have to correctly print handle the undefined value. Driving a wire twice in a single module is one example where the wire will have an undefined value assigned to it instead. As undefined behaviour is directly supported by simulators, semantically correct Verilog can still contain wires that are driven from multiple places, or even divided by 0, as the simulator should handle all of these with an undefined value. Accessing bits from a wire that are out of range also results in undefined values for those bits.

Semantically correct Verilog is not context-free like syntactically correct Verilog, as one needs to keep track of all the wires and variables that are in scope at any one time. This cannot be defined by having production rules that only have one element on the left hand side, as the context needs to be specified as well containing all the current modules, wires and variables that have been defined.

Context-sensitive deterministic Verilog

Deterministic Verilog is also a subset of syntactically correct Verilog. It is not a subset of semantically correct Verilog, as one can construct Verilog designs where the output is non-deterministic and simulator or synthesiser dependent. There is, however, a difference between determinism in synthesisers and determinism in simulators. Non deterministic behaviour in simulators appears because of differences in how statements are scheduled, as the simulator has to deal with inherently parallel constructs and sequence them in some way so that they can run sequentially on a processor. Synthesisers, on the other hand, behave non-deterministically when undefined behaviour is present in the Verilog. This is because undefined values are treated like “don’t cares” and the synthesiser can assign a 1 or a 0 to that bit. This is outlined in detail in Section 2.1.5.

Deterministic Verilog is also context-sensitive, as it is possible to construct a linearly bounded NDTM that evaluates all the possible outputs that a piece of Verilog code can produce and ensures that there is only one. At the same time, it could also detect if there is any implementation defined behaviour present. This has also been attempted in practice by creating a formal executable semantics of Verilog [28] which can then be used to check the deterministic nature of a piece of Verilog code. However, this is only feasible for small Verilog modules and the formal semantics do not cover the whole synthesisable Verilog subset. Alternatively, one can restrict the Verilog syntax to constructs that are known to be deterministic and limit the generated Verilog to adhere to the structure of an acyclic circuit, which guarantees that there can only be one output for every input to the module.

2.1.2 Simulation

Simulation is a way to execute the design before placing it on hardware, in order to check the correctness of the design and analyse how specific waveforms will interact with the design. Simulators often support useful Verilog constructs which cannot be synthesized, such as system tasks or the tri-state wire, that cannot be modelled correctly in hardware. These can be used in a test bench to test that the synthesisable design works. Simulators often compile the Verilog to an intermediate form, such as Icarus Verilog [29], but can also transform the Verlog into optimal C++ code instead which can then be compiled, such as Verilator [30].

Simulators that implement Verilog have to follow the IEEE 1364-2005 standard [26], which was the last standard before Verilog was combined with SystemVerilog in the IEEE

1800-2017 standard [31]. The IEEE 1364-2005 standard will be followed and specifies the Verilog syntax and semantics and defines the constructs that lead to undefined behaviour or non-deterministic behaviour.

As mentioned earlier, undefined behaviour should be handled properly by simulators, and therefore should not result in non-deterministic output, but instead in undefined bits wherever the undefined behaviour occurred. However, non-determinism is also a property of Verilog simulators. To simulate inherently parallel hardware, simulators have to run all the continuous assignments and always blocks in parallel, in order to emulate the layout in hardware. This can lead to race conditions between different updates to wires, and therefore could result in two different values. Section 2.1.5 discusses the non-determinism in simulators further.

2.1.3 Synthesis

Logic synthesis is the transformation of higher level code that does not have a direct translation to hardware, into lower level components that can be modelled directly by using an FPGA or ASIC [22]. Synthesisers such as Yosys, Vivado or Quartus can synthesize higher level Verilog to a lower level model called a netlist, which is a representation of the lower level components. The format of the netlist can vary widely, but for analysis purposes, these tools allow the netlist to be expressed in Verilog again, by using an extremely limited set of features. The netlist is tailored specifically to the type of hardware that it will eventually be placed on, meaning that different synthesisers will make different assumptions about what operations the hardware supports natively and which operations have to expressed in the netlist separately. Finally, to implement the netlist in hardware, the "Place & Route" maps the netlist exactly onto the hardware and outputs a bit stream that will configure the FPGA correctly.

An example of the synthesis of a Verilog module is given below, where the behavioural Verilog code is given in Listing 2.2. This is a simple behavioural circuit with an always block that assigns the output to the input at every clock edge using a non-blocking assignment. The output `y` is actually defined as a wire, and it is illegal to assign anything to a wire inside of an always or `initial` block. This should be defined as a `reg` to make the code work properly. However, it still synthesises under Yosys, which only outputs a warning that a `wire` is being assigned to in an always block.

```verilog
module top(clk, x, y);
   output reg y = 0;
   input clk, x;
   always @(posedge clk)
     y <= x;
endmodule
```

Listing 2.2: Manually written behavioural Verilog code.

After synthesising the Verilog module in Listing 2.2, the representation of the netlist generated by Yosys can also be shown in Verilog which can be seen in Listing 2.3. It shows the logical representation of the behavioural circuit defined in Listing 2.2 which can be directly mapped to hardware. First of all, the wires that are passed to the module are declared. As the always block synthesises to a flip-flop, it is represented by a module called `FDRE` with `clk` connected to the clock input in the flip-flop `C`. The input `x` is assigned to the input of the flip-flop `D` and `y` is assigned to the output of the flip-flop `Q`, which is how we would expect to implement the always block in hardware. Finally, the initial value for the output is set by a parameter `INIT` which is set to 0, and the enable `CE` and reset `R` are set to 1 and 0 respectively.

```verilog
// Implementation of flip-flop for simulation
module FDRE #(parameter INIT = 1'b0) (Q, C, CE, D, R);
   output Q;
   input  C, CE, D, R;
   wire   Q;
   reg    q_out;
   initial q_out = INIT;
   assign Q = q_out;
   always @(posedge C)  // Flip-flop is simulated with always block
     if (R) q_out <=  0;
     else if (CE) q_out <=  D;
endmodule

module top(clk, x, y);
  input clk, x;
  output y;
  FDRE #(.INIT(1'h0)) _0_   // Flip-flop instead of always block
    (.C(clk), .CE(1'h1), .D(x), .Q(y), .R(1'h0));  // Assign x to y
endmodule
```

Listing 2.3: Generated Verilog netlist by Yosys.

The `FDRE` module is normally not provided in the synthesised output, because it is assumed that the FPGA natively supports it. However, to perform any kind of simulation or formal verification on the Verilog netlist, the implementation of the `FDRE` module has to be provided to the simulator or formal verification tool. The `FDRE` module was therefore implemented by using an always block to simulate the flip-flop.

2.1.4 Formal Verification

Formal verification is the last set of tools that will be discussed, as these are used by VeriFuzz to determine the equivalence of the original design compared to the synthesised netlist. Formal verification tools can prove specific properties about hardware, which will always hold, no matter what inputs are given to the hardware. This section will focus on formal equivalence checking, which is a possible application of formal verification. Therefore, the property that should hold is that the outputs of both modules are always the same no matter the inputs.

```
module original(a, b, out);
   input a, b;
   output out;
   assign out = ~((~a & ~b) | c);
endmodule
```

(a) Original design.

```
module synthesised(a, b, c, out);
   input a, b, c;
   output out;
   assign _0_ = b | a;
   assign out = _0_ & ~(c);
endmodule
```

(b) Synthesised design (generated by Yosys).

Listing 2.4: Example of original and synthesised design which should be equivalent.

Synthesis performs many optimisations and transformations when mapping the original design to hardware, in order to fit onto the device that is being targeted. One example is the module shown in Listing 2.4a which synthesises to the module shown in Listing 2.4b. The bodies of both modules show a simple continuous assignment of the input to the output. The equivalence of those modules can therefore be expressed in logic

$$\neg((\neg a \land \neg b) \lor c) \iff (a \lor b) \land \neg c.$$

The above can be proven using De Morgan's laws, which state that

$$\begin{aligned} P \land Q &\iff \neg(\neg P \lor \neg Q), \\ P \lor Q &\iff \neg(\neg P \land \neg Q). \end{aligned}$$

Therefore, the equivalence can be shown by the following

$$\begin{aligned} \neg((\neg a \land \neg b) \lor c) &= \neg(\neg a \land \neg b) \land \neg c \\ &= (a \lor b) \land \neg c. \end{aligned}$$

As we can show that the two logical expressions are equivalent, we can also say that the initial design and the synthesised design shown in Listing 2.4 are equivalent.

This can also be done automatically using a satisfiability modulo theories (SMT) solver because the equivalence problem can be transformed into a satisfiability (SAT) problem. Given a logical expression, the SAT problem determines if there is a possible input such that the logic expression is true and therefore satisfiable. This can be done to prove equivalence between two logical expressions as well. The simplest way to do this is by first transforming the problem into a miter circuit shown in Figure 2.2. This circuit will output 0 if the two circuits are equal for the current inputs, and 1 if the outputs do not match. Therefore, to prove equivalence, we can instead prove that the miter circuit is unsatisfiable (UNSAT), which in turn proves that the two circuits give equivalent outputs for all possible inputs.

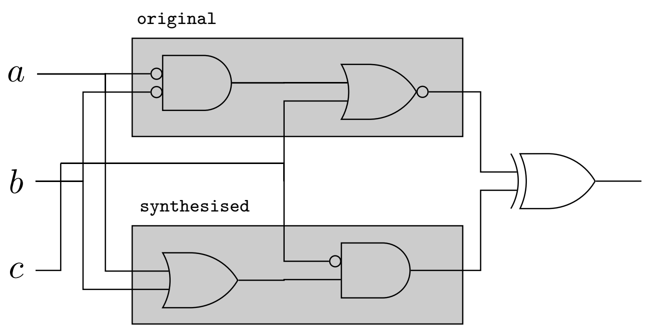


Figure 2.2: Miter circuit of the design and synthesised net list.

To be able to use an SMT solver, the miter circuit has to be expressed in logic which is shown below

$$(\neg((\neg a \land \neg b) \lor c)) \oplus ((a \lor b) \land \neg c),$$

and can then be input to the SMT solver. It will then identify an input that satisfies the equation, and if none can be found, the SMT solver will return that the equation is unsatisfiable. That means that both logical expressions are equivalent and therefore also means that both modules are equivalent.

For this to work, Verilog first has to be converted to logical expressions before an SMT solver can be used to find the solution to the Sat problem. There are a few tools that can do this, and can convert Verilog to SMT-LIBv2 [32] so that any SMT solver can be used to prove properties about the circuit. SMT solvers also often support temporal logic [33], which allows them to prove properties about sequential circuits as well.

However, there are also specialised formal verification tools that were developed specifically to work with hardware descriptions in various formats and can prove properties without first having to convert the circuit to a language that an SMT solver accepts. One such example is ABC [34], which is a tool specifically developed for sequential synthesis and verification. It uses Sat solving to prove the equivalence of miter circuits, but also tries to simplify the circuit into a functionally reduced graph so that the proof is as efficient as possible. ABC is therefore used by VeriFuzz to prove the equivalence of the original design and the synthesised netlist.

2.1.5 Determinism

Deterministic behaviour is an important property that has to be taken into account when comparing the outputs of different tools to check if one tool has a bug. If the outputs of the tools do not match, but non-deterministic behaviour was present in the original design, then it might still be the case that all the tools are conforming to the standard. Determinism manifests itself in different ways in simulators and synthesisers.

In synthesisers, determinism refers to the synthesised netlist, which should be equivalent to the design no matter what tool was used to do the synthesis. However, non-determinism can be introduced into the synthesised net list if undefined behaviour is present in the design. Undefined behaviour in Verilog is quite common and can be represented by the valued `x` in simulators. However, this is not possible in real hardware as wires can only be a 1 or a 0, but not `x`. Therefore, if undefined behaviour is present in the design, the tool can choose to replace that undefined behaviour by either a 0 or a 1, as it can be treated as a "don't care". Two synthesisers that are given the same design containing undefined behaviour might output netlists that are not equivalent to each other or the original design, as the synthesisers can assign anything they want to the bits that are undefined. Therefore, the module shown in Listing 2.5 is valid but the output could be set to either a 1 or a 0 depending on the synthesiser used. Therefore, any constructs that results in undefined behaviour cannot be used when testing the outputs of synthesisers.

```
module top(out);
   output out;
   assign out = 1'bx;
endmodule
```

Listing 2.5: Non-determinism example in synthesisers.

In simulators non-determinism manifests itself in a different way, as undefined behaviour will be represented by `x`. Therefore, a design that contains undefined behaviour needs to give the exact same output no matter what tool was used to simulate it. However, because of the way that Verilog events are defined in the standard, there can be possible race conditions that give rise to non-determinism in simulators. This is caused by the scheduling system in Verilog. In Verilog events are placed on different queues depending on when they need to be executed, where events are often updates to variables or wire values. The following queues present in Verilog are listed in order of execution, meaning events from a queue will not execute until all the events from the previous queue have finished.

Active queue contains all the events that should be currently executed.

Inactive queue is a special queue comprises statements that were scheduled with a `#0` delay.

Nonblocking assignment queue contains all the nonblocking assignment updates.

Postponed queue which contains all the events that need to occur happen at the end of the time step, such as the `$strobe` function, which prints out the values of variables at the end of the current time step.

However, Verilog is designed so that if a current event is being processed from a queue, it can be placed back onto the queue and paused at any time so that a different event in the same queue can get processed. That means that there might be race conditions if events depend on each other and are in the same queue. One example of such a race condition is the module shown in Listing 2.6, which shows a module that will print the value of `p` to the screen. However, printing a 1 or a 0 would be valid. To demonstrate why this is the case, the Verilog code in the module will be explained line by line. First, a continuous assignment is created between `p` and `q`, meaning that anytime `q` is updated, an event is pushed to the active queue which updates the value of `p` to be the same as `q`. Then, in the initial block, the event of setting `q` to 1 is pushed to the active queue. As a delay is encountered and there are no more initial blocks in the module, the events in the active queue are executed, therefore `q` is set to 1. Subsequently, another event is pushed to the event queue which should update `p` to 1 as well, as `q` changed.

```
module main;
   wire p;
   reg  q;
   assign p = q;
   initial begin
      q = 1;
      #1 q = 0;
      $display(p);
   end
endmodule
```

Listing 2.6: Non-determinism example in simulators.

Because there are no more events in any of the queues, the simulator moves on to the next time step. There, the event of setting `q` to 0 is pushed to the active queue and is executed, which then pushes the event setting `p` to 0 and the print statement `$display` to the queue. As the execution in the same queue is non-deterministic, the print could either execute before `p` is set to 0 or after that, and therefore either a 1 or 0 could be printed.

The determinism described in simulators does not apply to synthesisers, as the standard says that delays can be ignored and `$display` is not a valid task in synthesis. Therefore, the synthesiser should behave deterministically, as the implementation will just be a wire between `p` and `q`, meaning that no scheduling issues arise.

2.2 Fuzzing

The term “fuzzing” was introduced by Miller et al. [35] and stood for generating a random input that the program under test (PUT) did not expect. The test was positive if the PUT did not crash once it had received and processed the input. This simple fuzzing method revealed many different bugs in common UNIX tools such as emacs, vi, csh, spell and uniq, with diverse types of errors ranging from array or pointer manipulation bugs to race conditions. However, as a result of only inputting random bytes, these bugs were mainly caused by the early stages of the programs execution and failed to address more complex errors. As fuzz testing methods developed, more advanced generation and evaluation methods were being used to achieve better results.

Fuzzing is therefore a way to test software automatically, by passing randomly generated data to the program and verifying the correct processing of the input. Correct processing of input data could simply mean that the program does not crash when it is passed the data, but it may also mean that the program executes in the intended behaviour. The latter can be achieved by restricting the input to a valid subset that the program will accept and devising a method to find out if the execution was correct based on that input. This gives information about the quality of the program itself rather than purely about the quality of the parser and is therefore vital to identify bugs located deeper in the program.

This notion of fuzz testing can be applied to compilers, whereby a large number of random programs are automatically generated and given to the compiler, in order to discover bugs. CSmith [36] is one example of such a fuzzing tool for C compilers and has been applied with great success to mainstream C compilers. It has led to the discovery of hundreds of bugs in compilers such as GCC and Clang and has even found bugs in verified compilers such as CompCert [37]. Bugs in the latter occurred in unverified parts of the compiler and motivated their verification. Apart from CSmith, a considerable amount of work has been carried out on fuzzing different kinds of compilers using various different fuzzing methods, such as a JavaScript fuzzer [38], a Rust typechecker fuzzer [39] and a DOM fuzzer [40], all of which identified many bugs in the compilers they were testing. This confirms that fuzzing is an effective method to test compilers. However, generating random models can also be used to test protocols like TLS [41] or NFC [42], where the general structure of the random input messages is correct so that it tests the implementation of the protocols more deeply.

Although published work on testing hardware simulators and synthesisers using fuzzing techniques is limited, advances in fuzz testing compilers can be applied to test these tools. This section will cover the main types of fuzzers in Section 2.2.1 and what they are commonly used for. It will then focus on different types of input generation and evaluation techniques that are specifically used when fuzzing compilers and that have been found to be effective in other languages.

2.2.1 Fuzzer Types

There are three main fuzzing techniques that greatly affect the different stages in a fuzzer [43]. VeriFuzz falls into the category of a black-box fuzzer, however, there are also white-box fuzzers and grey-box fuzzers.

Black-box fuzzing is at the simplest but often more general level. The term “black-box” refers to testing a program without assumptions or knowledge about the inner workings of the PUT [44], [45]. In most cases fuzzers that test specific protocols or grammars will be black-box fuzzers and are normally used to test various implementations of the protocol or compilers that accept that grammar. The fuzzers should therefore not make any assumptions about the implementation, as corner cases in the specification may never be reached. If the fuzzer only focuses on the specification, it also means that it should be able to test any implementation quite well. One such example is CSmith, which generates random C programs used to fuzz existing C compilers such as GCC, Clang or CompCert. CSmith does not rely on the internal workings of the compiler, and can therefore be used to test any C compiler.

White-box fuzzing is at the other extreme and leverages symbolic execution and dynamic test generation to create tests that are specific to the PUT [46], [47]. Instead of providing input values to the program, symbolic execution [48] assigns symbols to the inputs and passes through equations in terms of those symbols. In the case of conditional statements, each one corresponds to a tree of different possible execution paths subject to constraints which can be recorded. This information can be used to generate dynamic tests covering all the possible branches, even rare scenarios unlikely to have been explored with random black-box testing. Listing 2.7 illustrates this point, because if the values generated for `a` were random, there would only be a $1/2^{26}$ chance that the code path would be executed, assuming `unsigned` is 32 bits wide, whereas white-box fuzzing would generate tests for all possible execution paths.

```
int func(unsigned a) {
    if (a < 64) { abort(); /* error */ }
    return 0;
}
```

Listing 2.7: Error unlikely to be hit with random black-box testing, but white-box testing will generate tests for all possible branches.

Grey-box fuzzing is a third test alternative in between white-box and black-box fuzzing. White-box fuzzing is time consuming and computationally expensive, because it requires in depth analysis of the PUT which may be infeasible for large programs. Instead, a more lightweight analysis of the PUT can be sufficient, such as performing static analysis on the code, or introducing some instrumentation to improve the test case generation. An example of a grey-box fuzzer is american fuzzy lop (AFL) [49] which uses compile-time instrumentation and genetic algorithms to discover test cases it deems interesting and that are likely to change the internal state of the binary. Another example is VUzzer [50] which uses static and dynamic analysis of the PUT in order to produce the best test-cases with the most coverage.

VeriFuzz is a black-box fuzzer, because it generates a Verilog file independently of the simulator or synthesiser being tested. It is able to test any Verilog synthesis tool based on the assumption that the tool accepts Verilog and produces the synthesised logic in as well Verilog. It is therefore not possible to modify the source of the simulators and synthesisers being tested to introduce instrumentation. The quality of the fuzzer only depends on the quality of the randomly generated Verilog code, as well as how well the fuzzer performs at detecting a bug once the generated code has been run through the PUT.

2.2.2 Fuzzing Steps

Algorithm 2.1: General fuzzing algorithm.

```
  Input  : $\mathcal{C}$                                    // Configuration
  Output : $\mathcal{B}$                                    // Bugs found
1 $\mathcal{B} \leftarrow \varnothing$, $\mathcal{R} \leftarrow \varnothing$;        // Initialise bugs and runtime information
2 repeat
3     $g \leftarrow$ InputGen($\mathcal{C}$);
4     $b$, $\mathcal{R} \leftarrow$ Eval($\mathcal{C}$, $g$);
5     $\mathcal{C} \leftarrow$ Update($\mathcal{C}$, $b$, $\mathcal{R}$);
6     $\mathcal{B} \leftarrow \mathcal{B} \cup \{b\}$;
7 until StopCondition($\mathcal{C}$);
```

Most fuzzers follow a similar general structure which can be seen in Algorithm 2.1, which VeriFuzz also follows. The input to a fuzzer is usually a configuration $\mathcal{C}$ and the output is a set of bugs $\mathcal{B}$ that were found in all the fuzz runs. The main fuzz loop performs all the necessary steps to test the PUT and records any bugs that were found. There are three main steps that can be identified in a fuzz run and which are looped over.

1. Input generation
2. Evaluation
3. Configuration update

First, the input is generated using the `InputGen` function, which will then be given to the PUT. The `Eval` function then executes the PUT with the generated input and checks the output is correcting. Any bug b that is found during evaluation is returned, including runtime information $\mathcal{R}$ that may have been recorded during the execution of the PUT. Finally, the configuration is updated using the `Update` function so that future iterations of the fuzz runs are effective in that they identify more diverse bugs or test a feature more thoroughly. At the end of the loop, the set of all the bugs found is updated.

Input Generation

The input generation step of the fuzzer is important, because it determines the success rate of the fuzzing run. The two main methods used are input mutation and random input generation. The former accepts a valid input and mutates it in random ways to generate the different inputs to the PUT. It is often used to fuzz protocols or binaries where the input may not be specified but examples are available. The examples can then be used to generate new test cases, which can be complemented with instrumentation which is what AFL does. If the input is well defined by a grammar, the input is normally generated randomly from a seed. Therefore random input generation is frequently the generation method of choice when testing compilers, as it is difficult to perform valid mutations on grammar and simpler to generate random programs from scratch using the language's grammar.

One such example is CSmith, which generates C code procedurally by keeping a state of variables and functions in scope at any point in time. It has separate probabilities to generate different constructs at every step, such as declaring a variable or calling a new function. If it decides to call a function that is not currently in scope, it will generate the missing function first in the same way before finishing the call to the function and then proceed. This ensures that the generated C code is always correct and expressions will either include variables declared earlier or declare them before use. The goal of CSmith is to generate large random C files, because files with 8193–16384 lines were shown to produce the highest chance of identifying bugs in a specified amount of time. If CSmith generated programs of a different size, less bugs were found in the same amount of time; smaller test cases did not contain as many interesting interactions, and larger files took much longer to generate and compile, therefore not being as time efficient.

Evaluation

The evaluation step not only runs the PUT with the generated input, but also evaluates the output to detect any bugs. The most primitive way to evaluate the output of a program is to check if the PUT aborts and performs a core dump, which is rarely the expected behaviour, as even a program that is passed completely random bytes is expected to fail gracefully. More interesting bugs can also be found using other evaluation methods, such as differential testing [51], where the outputs of multiple programs are compared or stack based hashing [43], where the expected call-stack is predicted or evaluated beforehand and is compared to the actual call-stack. Finally, the expected outcome can sometimes be predicted before hand and compared to the actual output. This step will also return runtime information about the execution. For example, AFL, will analyse the binary during execution and will report interesting inputs that set the binary into a rare state, which often results in a bug.

Configuration Update

To ensure that every fuzz run improves on the last one, the configuration is updated based on the current configuration and the outcome of the run. These vary wildly between different fuzzers, as grey-box fuzzers will tend to update the configuration in many more ways than black-box fuzzers, as more run-time information is available. However, different ways to update the configuration file are applied with the intention of diversifying the tests and widen the scope of bugs found. One such technique is swarm testing [52], which randomises the configuration file for a specific amount of time and can therefore find a larger variety of bugs.

2.2.3 Main Fuzzing Techniques

This section will focus on various fuzzing techniques that were implemented in VeriFuzz. To compare the outputs and discover bugs, a variation on differential testing [51] was used. Swarm testing was used to update the configuration after a set amount of runs to diversify the bugs that were found. Finally, test-case reduction will be described, as it is implemented in VeriFuzz to reduce the size of the random designs to a test-case that could be included in a bug report and clearly show cases the bug. Often times, however, the reduced test-case has to be reduced further manually to achieve that.

Differential Testing

Differential testing [51] can be used to evaluate any input, given that there are at least three different compilers for the language that is being fuzzed. Given a valid input, a bug can be found in one of the three compilers being tested by passing the input to each compiler and checking whether the output is identical. If there are any discrepancies, the compiler that produced the discrepancy is assumed to contain a bug.

This evaluation technique is the most common among language fuzzers, as it is likely that there are at least three different compilers available. Even if that is not the case, differential testing can also be used with the same compiler at different optimisation levels to find bugs in optimisations that the compiler performs. This approach does not depend on a specific generation method like other evaluation methods, and therefore can be used in any compiler fuzzer effectively. It was chosen for VeriFuzz as well, because random Verilog generation allows for full control over undefined behaviour.

However, in VeriFuzz, a slight variation of differential testing is used. For the comparison of synthesiser output, the synthesised netlist is checked for equivalence against the original instead of comparing it against the other synthesisers. If they are found not to be equivalent, there has to be a bug in the synthesiser. This is due to the fact that the generated Verilog code is deterministic and therefore there is only exactly one possible output for each input. If one of the outputs does not match, the logic of the netlist does not match the logic of the design, and therefore the synthesiser must have a bug. If that were not case and undefined behaviour was allowed, a simulator would have to be used to check that the discrepancies between the design and the netlist only happen when there are undefined bits in the simulation of the design.

However, testing simulators could be done using differential testing if multiple simulators were supported, but having equivalent synthesised netlists available in Verilog allows for the inverse of differential testing to be used. As the netlists are equivalent, simulating both with random inputs should always result in the same output and if there are discrepancies it follows that there must be a bug in the simulator. This is similar to a technique called equivalence modulo inputs (EMI) [53], which is described further in Section 2.2.5.

Swarm Testing

Swarm testing [52] is a useful technique that can be applied to find a larger variety of bugs when fuzz testing. The idea behind swarm testing is that targeted testing a few features is more effective than testing all the features at once and as a consequence will result in identifying a more diverse set of bugs. Instead of always generating a random program by allowing all the constructs in that grammar, multiple different random programs are generated, each with some constructs randomly disabled. The reasoning behind this is that some constructs can limit how thoroughly other constructs are tested and may prevent some bugs from being found. In addition to that, as some features are disabled, it allows for deeper testing of the features that are enabled by trying out more combinations between that subset of features.

2.2.4 Test-case Reduction

Finally, test-case reduction is a general technique that is included in fuzzers to minimise failures that were found, so that these can be analysed further and reported. The main technique that is used for this is delta-debugging [54], which is language agnostic automatic debugging technique, which can therefore be used for the reduction of random test-cases as well. There are various different implementations of delta-debugging, such as Delta [55] which is a straight forward implementation of delta-debugging targeting C, or CReduce [56] which is also a reducer for C and was written as part of CSmith. The latter uses various different reduction steps such as text-based delta-debugging like Delta, but also includes various AST reduction steps.

The general idea behind delta-debugging is that it performs a search on the code by splitting up the code into different regions and keeping regions that show the unexpected behaviour. Therefore, Running the delta-debugging tool multiple times with regions of decreasing size should eventually identify the lines of code that are causing the unexpected behaviour. This is exactly the algorithm that Delta implements, whereas CReduce also includes a search on the AST and leverages formal verification to only check code that does not contain any undefined behaviour in C.

CReduce is heavily specialised for C and can therefore not be used to reduce Verilog. On the other hand, as Delta is quite general, it can be rewritten to accept and reduce Verilog. However, It is shown in Section 5.5 that even though it manages to reduce the Verilog to quite a minimal representation, it takes so long that it is not practical to use. Therefore, a custom reducer was implemented in VeriFuzz which performs a greedy binary

```
int func1(int a, int b) {
    return a;
}
```

(a) Original piece of code.

```
int func2(int a, int b) {
    if (!b) { return b; }
    return a;
}
```

(b) Equivalent code given $b \neq 0$.

Listing 2.8: Example of EMI applied to a small function.

search on the AST to find a small test-case that shows the same unexpected behaviour and it is shown that it reduces the designs to a similar size as Delta, but in a fraction of the time.

2.2.5 Alternative Fuzzing Techniques

Many alternative fuzzing techniques have been shown to be effective for testing compilers. One example is skeletal program enumeration [57], which tests a small code template thoroughly, or EMI testing which generates multiple programs that are equivalent when executed with specific inputs. These have been considered for VeriFuzz, but not implemented, as they were deemed too complex compared to the simpler solution of comparing the original design to the netlist.

Skeletal Program Enumeration

Skeletal program enumeration is a different way to generate test cases for a compiler. Instead of procedurally forming large C files that are then compiled and tested, it generates a small template of a program and then enumerates all possible variable patterns that can appear in the template. The motivation is that reduced test cases in the GCC bug reports only contain 30 lines of code on average [58]. Their reasoning is that testing all possible combinations of a randomly generated template will be more likely to find bugs than randomly generating a large program and hoping to find a bug that can then be reduced.

The reason this technique was not chosen for VeriFuzz is because skeletal program enumeration relies on the testing of many different variations of a given template before a bug may be found. It may also be the case that the given template does not actually have a bug in any possible combination. As synthesis is highly time consuming, it may not be feasible to test all these possible combinations effectively. It is therefore better to generate Verilog randomly and try and find a bug which can then be reduced.

Equivalence Modulo Inputs

Le et al. [53] showed that mutation could be used to test compilers in a reliable way using EMI testing. Instead of the usual random generation method used by CSmith and many other compiler fuzzers, EMI works by generating multiple test-cases $\mathcal{P}$ from an exiting program, that are equivalent under specific inputs $\mathcal{I}$. These can all be run through the same compiler and evaluated with the inputs $\mathcal{I}$. As all the test-cases in $\mathcal{P}$ should be equivalent under those inputs, differences in the outputs given the inputs $\mathcal{I}$, will reveal a bug in the compiler. An example of EMI is shown in Listing 2.8, where `func1` is the original function and `func2` is the mutated function. `func2` is equivalent to `func1` if `b` is never set to 0, as the everything in the if-statement effectively becomes dead code. However, the reason this technique works is that given `func2`, the compiler that is being tested cannot make any assumptions about the if-statement, and therefore has to compile the whole function correctly. When checking for any bugs in the compiler, all valid inputs where `b` is not 0 can be checked as they should all give the same result.

The benefit of this technique is that only one compiler is needed, which is crucial if less than three compilers necessary for differential testing are available. In addition to that, it is also less error prone, as there might be bugs in two of the compilers, leading to three different results, making it difficult to find out which compiler is correct. In EMI, this is not possible, as there is always the original program which gives the correct output, and all the other programs can be evaluated by comparison.

This technique was not chosen for VeriFuzz, because to perform these random mutations on existing code, all of the synthesisable subset of Verilog would have to be supported instead of only a smallest subset of deterministic Verilog. If mutations are performed on existing code, it is extremely likely that undefined behaviour will be present in the design, as it is quite difficult to avoid in practice. As a consequence formal verification on its own could not be used to check if the programs $\mathcal{P}$ are equivalent to the original, as the synthesiser can implement it in any way that they want to. In contrast, if deterministic Verilog code was generated and mutated using EMI afterwards, it would be simpler to use differential testing with formal verification instead.

2.3 VlogHammer

VlogHammer [25] is a differential tester targeting simulators and synthesisers, both commercial and open source. It was initially built to test the open source Verilog synthesis suite called Yosys [24], which was written by Clifford Wolf, but due of the nature of differential testing, it found bugs in other tools as well. It now supports many common commercial Verilog synthesisers such as Quartus [59], Xilinx ISE (XST) [60] and Vivado [61], as well as simulators such as Xilinx Isim and Xsim which come with ISE and Vivado respectively, Modelsim which is included in Quartus, and finally an open source simulator called Icarus Verilog [29]. There is a distinction between synthesis and simulation in Verilog simulators and the aim of VlogHammer is to test the correctness of synthesisers by using simulators to check the output.

2.3.1 Synthesis and Simulation

VlogHammer aims to test the correctness of synthesisers. The main workflow used by VlogHammer is to first generate many small random Verilog modules. These are then passed to each of the synthesisers to be tested and synthesized by each of them. If there are any failures during this process, the failing tools will be noted and the failing files will be set aside to be included in the final report. However, this should be extremely unlikely, as in contrast to software compilers, these synthesisers will try to output a synthesised file, even with unsound Verilog files. For example in XST, if there are conflicting parameters or attributes, the tool will try to solve the conflicts and figure out what the user meant instead of outputting an error [62].

Synthesis

Once the synthesized files are generated, they have to be made architecture independent to be compared to each other, as each synthesis tool has a different target platform. These architecture differences are expressed in the synthesised Verilog files by specific module instantiations that the target platform supports natively. As the author of VlogHammer is also the author of Yosys, the synthesised files are made architecture independent by converting them into the Yosys netlist format called "ilang". Extra Verilog source files are supplied for the conversion which provide Verilog models of every module that might be instantiated in the Verilog netlists for the different hardware modules, which the target architecture natively supports.

Once all the "ilang" files are generated for all the synthesised modules for the synthesisers, they can finally be compared against each other for logical equivalence, by using a form of differential testing. The original Verilog module that was passed to all the synthesisers is also compared against the output after synthesis, to ensure equivalence. Yosys provides bindings to the ABC formal verification tool, as well as conversion to SMT-LIBv2 [32], a language standard that is supported by many existing SMT solvers. An example query is shown in Listing 2.9, which proves that `y1`, the output of one module, is equivalent and therefore always equal to `y2`, the output of another module, independent of what is passed to the inputs `a1`, `a2`, `a3`. The `-show` option selects the signals that should be available to the command, which in this case contains all the signals in the module, and the `-prove y1 y2` command shows that signal `y1` is always the same as signal `y2`.

```
sat -show a1,a2,a3,y1,y2 -prove y1 y2 verilog_module
```

Listing 2.9: Example SAT query in Yosys to prove equality between y1 and y2.

The inputs are passed to two modules called `rtl_module` and `yosys_module`, where the former is the originally generated verilog code, and the latter is the synthesised verilog code by the Yosys synthesis tool. They output `y1` and `y2` respectively which have to be shown to always be equal by the SAT solver, defined in the top level module shown in Listing 2.10. By proving that the outputs of the two modules will always be identical, it is demonstrated that the modules are equivalent, therefore proving that the synthesised output by Yosys is correct.

```verilog
module verilog_module(a1, a2, a3, y1, y2);
   input [3:0] a1;
   input [3:0] a2;
   input [3:0] a3;
   output [89:0] y1, y2;
   rtl_module rtl_module(.a1(a1), .a2(a2), .a3(a3), .y(y1));
   yosys_module yosys_module(.a1(a1), .a2(a2), .a3(a3), .y(y2));
endmodule;
```

Listing 2.10: Top level module that is read by the sat query and is used to define the problem for the SAT solver.

Simulation

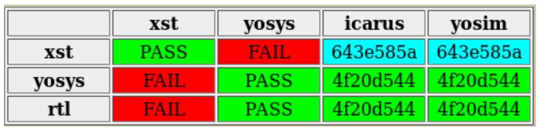
As the comparison step between the synthesised Verilog modules uses formal proofs to show that they are equivalent or not, there can never be any false positives. If the synthesised modules are found to be equivalent, they will always have the same output for all possible inputs to the module. It is then likely that both of those synthesisers are correct and generate the correct code after synthesising the module, as it is improbable that they both generate identical wrong code. In addition to that, if the synthesised code is shown to be equivalent to the input module, it is certain that the synthesis step was correct.

If, however, the tool finds that the two modules are not equivalent, it does not necessarily mean that one of the modules generated wrong code, because false negatives are possible. In Verilog, there are expressions that will evaluate to an undefined value. Although this can be handled by simulators, as they can assign `x` to values that are undefined and continue simulation, synthesisers cannot as in hardware, wires or registers have an undefined value. Instead, any undefined value during synthesis is interpreted as an undefined signal, which means that its value is implementation defined and can change in between synthesisers.

To detect these false negatives, the initial Verilog modules together with the synthesised modules are simulated with a test bench that randomly assigns values to the inputs and records the outputs. In addition to random inputs, the counterexamples found by the SAT solver are also added to the test bench to trigger the edge case that produced the inequality between the two synthesised modules. All the outputs of the simulator are hashed, and if there are any undefined bits in the output, they are masked. The hashes that were obtained from all the synthesised modules are then compared. If the modules were found to not be equal to each other, but the output hashes from the two simulators agree, then it was a false positive and there were undefined symbols in the design.

Evaluation

Finally, a table, which can be seen in Figure 2.3 is generated to get an overview of the failures and where they occurred, and this can be combined into a large report. The first column shows all the synthesisers that were tested, where **rtl** stands for the original design that was input to the synthesisers. The two next columns then display the same synthesisers that were tested and the results in each row show if the formal equivalence check passed or failed. For example, Figure 2.3 shows that XST failed the equivalence check with Yosys and the original design, but passed the equivalence check with itself. The two last columns contain the hashed results of running each synthesised netlist through a simulator with a test bench. In Figure 2.3, it can be seen that both icarus and yosim obtained the same output for each netlist, but that the netlist produced by XST differs to the other netlists and the original design.



	xst	yosys	icarus	yosim
xst	PASS	FAIL	643e585a	643e585a
yosys	FAIL	PASS	4f20d544	4f20d544
rtl	FAIL	PASS	4f20d544	4f20d544

Figure 2.3: Table included in report for test case failure in XST, generated by VlogHammer.

The way VlogHammer evaluates the different test runs is using differential testing, similar to how CSmith evaluates the outputs of their test runs. However, as the test cases generated by VlogHammer are fairly simple and small, it also uses a SAT solver to formally prove that the synthesised output is equivalent to the original Register Transfer Level (RTL) that was generated by VlogHammer. In addition to that, it also proves that the synthesised Verilog modules by all the synthesisers are also equivalent. This is shown in Figure 2.3, where XST and Yosys are compared to each other and to the original RTL. In this case, a bug was found in XST, as it is only equivalent to itself and not to the module synthesised by Yosys or the original RTL.

As the synthesised Verilog files are being proven to be equivalent, there cannot be any false positives, so if two modules are shown to be equal, they will behave exactly the same way for all possible inputs. However, the same cannot be said for false negatives. This is because there can be undefined behaviour inside the original Verilog module. As it is impossible to represent undefined signals in hardware, if any signals are undefined, the synthesiser can assume that they are "don't cares" and can assign any value to that wire. This implies that two modules could differ because of some undefined signals that have been assigned different values and that no bugs are present.

This is why simulators are used on top of the equivalence check, as they can mask out the undefined behaviour. Simulators, contrary to synthesisers, have to handle undefined behaviours by assigning `x` to the wire, meaning the value of the wire is not defined. Therefore, if the synthesised modules are both simulated using a test bench that contains the counter examples the SAT solver found and the output for both modules are identical when masking the `x` in the output, the conclusion can be made that it was indeed a false negative. If the results of the simulation are different, one can be sure that the modules are indeed different and that there is a bug in one of them.

In the case of XST in Figure 2.3, the results of the simulation are different to the other results, which means that VlogHammer did indeed find a bug.

2.3.2 Limitations

VlogHammer has successfully identified many bugs in all the synthesisers and simulators that were tested, however, it does have some limitations. First of all, VlogHammer does not support generating behavioural code, such as always and initial blocks. Instead, it focuses on generating correct Verilog expressions and testing those by only using continuous assignments. This limits the bugs that are found by VlogHammer to those that happen in expressions. However, VeriFuzz shows that there are bugs that only manifest themselves in specific constructs in behavioural Verilog that cannot be found by VlogHammer.

In addition to that, the generation step itself is also quite structured, only the expressions assigned to each wire are random. Therefore, it may not find bugs if they require a slightly different structure as the generation step does not allow for that.

Finally, VlogHammer is based on many different bash scripts that synthesise, simulate and verify the correctness of the synthesis or simulation. It relies on many Unix tools being installed to correctly execute. As it is fragmented into different bash scripts, it is difficult identify where specific files are created, and in what order the execution will happen.

Chapter 3

Project Specification

The aim of this project is to write a fuzzing tool called VeriFuzz that generates random Verilog to test different simulators and synthesisers. The motivation for this is to improve the quality of synthesisers and simulators, as these play a crucial role in generating correct hardware. Mistakes that are introduced by synthesisers are also especially hard to detect, so the designers have to put a lot of trust into the tools. Subsequently VeriFuzz will be run with the goal to extensively test the different simulators and synthesisers in order to verify that they simulate and synthesise the Verilog code correctly and create a correct netlist for all the random Verilog modules that are generated.

VeriFuzz is the library and command line tool that implements the random Verilog generation and the distribution of the code to the different simulators and synthesisers. It is designed to compare the outputs of the synthesised tools to the input design by using formal equivalence checking. If the outputs are not equivalent, the test-case is reduced to a minimal example that showcases the discovered bug. A waveform is generated from the minimal example which shows the discrepancy in how it was interpreted. The core contributions of this project are the following:

- Correct, deterministic, random Verilog generation.
- Interface to simulators and synthesisers
- Extensible library to support new Verilog generation, reduction techniques or the addition of new simulators and synthesisers
- Fuzz testing of the following synthesisers: Yosys [24], XST [60], Vivado [61] and Quartus [59]
- Fuzz testing of the Icarus Verilog [29] simulator
- Support for formal verification to compare synthesised netlist to the original design
- Test-case reduction once a bug is found to get a minimal test case exposing the bug

The requirements for each of these deliverables are described in the next sections.

3.1 Verilog Generation

The first core deliverable is to generate random, correct and deterministic Verilog. Two different methods were developed to generate random Verilog and compared against each other. The first method used a random DAG representing the randomly generated circuit to produce a deterministic circuit that could then be turned into Verilog. The second method used state-based generation similar to CSmith's generation of C to make sure that the generated Verilog is deterministic. The generated Verilog should be a subset of synthesisable Verilog, so that it can target synthesisers and simulators simultaneously. Below is an explanation of what is meant by random, correct and deterministic:

Random The fuzzer should generate different Verilog code at every fuzz run, which is seemingly random, however, it should still be deterministic. This is because runs should be able to be replayed if one has the seed that was used to generate the random Verilog, so that failures can be run again at will with the fuzzer and checked again easily by only using the seed. This is important as it enables the recreation of the whole internal state of the fuzzer which found the error, instead of parsing the generated Verilog that found the error and trying to recreate the internal state. This can be achieved by using a pseudo-random number generator instead of a fully random source, which uses a seed to generate a deterministic sequence of random numbers that can then generate the random Verilog. Therefore, if one has the seed, one can run the fuzz test in exactly the same way as it was run initially. One can then perform many manipulations on the internal representation of the Verilog code or the circuit it was generated from.

Correct The generated Verilog also has to be semantically correct with regards to the IEEE 2005 standard [26], which is the standard that is supported by most simulators and synthesisers. This means that wires should always be declared explicitly before they are used and should be declared to the right type, so that simulators or synthesisers cannot generate implementation specific behaviours. This is discussed further in Section.

Deterministic The generated code should also be deterministic, meaning that for all possible inputs to a module, there is only one possible output. This makes it possible to perform differential testing between the different synthesisers and simulators, as it would otherwise be possible to get different results for both simulators, while both having interpreted the Verilog code correctly. The way in which determinism is achieved in the first deliverable is by generating a random circuit in the form of a DAG, which will only have one possible output for any input.

However, because of this restriction, the fuzzer will never be able to find bugs that are caused by undefined behaviour in Verilog. As it is completely valid to introduce such undefined behaviour, and even though synthesisers can handle those bits as they wish, simulators still have to give the same output. Any bugs using undefined behaviour will therefore not be found.

Most of the synthesisable subset of Verilog can be generated by the state-based generation method in VeriFuzz. The generated Verilog programs will have the following features:

- module definitions with parameter definitions, inputs and outputs

```verilog
module top
// parameter declarations
#(parameter param = 5)
// input and output ports
(a, b, c, y);
   input a, b, c;
   output y;
endmodule
```

- module items, such as instantiations, continuous assignment, always blocks, initial blocks, parameter and local parameter declarations

```verilog
module top;
   // module instantiation
   modinst instance(.y(y), .clk(clk));
   // continuous assignment
   assign x = 1'b1;
   // always block
   always @(posedge clk);
   // initial block
   initial;
   // parameter declaration
   parameter param = 2;
   localparam local = 3;
endmodule
```

- most expressions, for example concatenation (`{1'b1, 1'b0}`), arithmetic operations (`1'b1 + 2'b1 ^ 1'b0`), ternary conditional operator (`x ? 1'b1 : 1'b0`)

- behavioural code in sequential always blocks

```verilog
always @(posedge clk) begin
   x <= 1'b1;
   y <= 1'b0;
end
```

- behavioural control flow such as if-else and for loops

```verilog
always @(posedge clk) begin
   if (x == 3'b2) x <= 3'b5;
   else x <= 3'b2;
   for (i = 0; i < 3; i = i + 1)
      x[i+2:i] <= 3'b1;
end
```

- declaration of wires and variables of any size, signed or unsigned

```
input wire signed [15:0] x;
output reg [2:0] y;
```

- bit selection from wires and variables

```
x[5:1] <= y[6:2] + z[19:15];
```

The DAG generation only supports a subset of those features, namely:

- module definitions, instantiations and continuous assignment
- most binary expressions such as gates and arithmetic operations
- declaration of wires and variables of any size

However, after the Verilog has been generated from the DAG, it can be modified further randomly and have behavioural code injected into some of the modules.

Some notable constructs that are not supported are functions and tasks, as implementation of these in synthesisers is quite straightforward. Tasks and functions cannot have any side effects, which means the synthesiser can always inline them before the Verilog is processed further. Therefore, it is expected that there are not too many bugs that can be found in those constructs. Furthermore, declarations of memories are also not supported. The plan is to add them in future versions of VeriFuzz.

3.2 Tool Support

The second deliverable is to support the main simulators, synthesisers and formal verification tools so that the simulators and synthesisers can be tested. For simulators, this means that given a test bench, all the simulators should return the same result after each of the inputs are passed to the module being tested. For synthesisers, this means that the synthesised netlist should be equivalent to the original design, given that there were no undefined values or no non-deterministic behaviour in the design. The comparison between the generated netlist and the original design can be done by converting the netlist back into Verilog and using formal property checking tools to prove the equivalence between the two.

As a requirement is that VeriFuzz should be extensible with more simulators and synthesisers, the implementation details of how the simulators and synthesisers are invoked has to be hidden in an interface that each simulator and synthesiser has to implement. Adding a new tool is then just as simple as implementing the required functions that the interface provides. The tool should then be ready to be tested.

3.2.1 Simulators

Checking that simulators behave in the same way as each other is done by using differential testing. This can be done in two separate ways. First of all, if many simulators are supported, differential testing can be used to find any discrepancies in the output of a simulator and classify it as a bug in one of those simulators. However, as multiple synthesisers are supported as well, a similar method of checking equivalence can be used for the simulators. The simulator can be run over the original design and over all the synthesised netlists using the same test bench, which means that the results should also be the same. This means that testing can be done with only one simulator being supported, but being run over all the synthesised files to see if it always generates the same results.

3.2.2 Synthesisers

Checking the equivalence of the synthesised netlists of different synthesisers can be quite difficult, as the formats the netlists are produced in are often different for each tool. On top of that, similar to how assembly cannot be directly compared with other assembly to check the equivalence, synthesisers execute many optimisation passes which will result in vastly different netlists that are still equivalent. In addition to that, synthesisers often target a specific FPGA supporting specific constructs like look-up table (LUT) or other hardware blocks. These are presented as primitives in the netlist, as the tool assumes that they are present. However, different tools will have many different modules that it assumes to be present in the hardware that it is targeting.

The solution to this is to convert the netlists back into simplified Verilog which connects wires in the same way the netlist does and instantiates modules for the primitives that are assumed to be present. Luckily, all tools that are being targeted for fuzzing support this directly, as it is often necessary to simulate the synthesised netlist to check that everything still works in the same way. This means that Verilog can either be generated as a netlist directly, which Yosys and Vivado support, or can be converted after the fact using a separate tool, which Quartus and XST support. Therefore, an interface can be defined such that it takes in Verilog, and returns synthesised Verilog that can then be compared to the original design.

3.2.3 Formal Property Checking Tools

Finally, to actually compare the synthesised Verilog to the original design, the fuzzer needs to be able to interact with formal property checking tools that can perform the equivalence checking. The equivalence checker should be able to take two designs and determine if there are any discrepancies.

This step should not give any false positives or false negatives, as only deterministic Verilog is generated, meaning the synthesised module has to be equivalent to the original design. This means that the result that the equivalence checker gives should directly show if there is a bug in the synthesiser or not.

3.3 Test-case Reduction & Evaluation

Once the fuzz runs have been performed, it is important to be able reduce them to a small test-case that can then be analysed to find the cause of the bug. Otherwise, the bugs that are found using the fuzzer will not be fixed by the maintainers of the simulators or synthesisers as they will not have time to debug randomly generated RTL. After the Verilog has been generated, it is useful to generate a report that has all the required information in it about the fuzz run that produced the error, such as the fuzzer configuration and seed that was used. Statistics about the run can also be useful, such as how long synthesis and the equivalence checking took. All these statistics are combined into an HTML file that can be viewed after all the runs have finished.

3.4 Command Line and Library API

The last deliverable is to produce a good command line tool and a library that is easy to use and can be extended to support more features. To provide such an API, it is useful to have a Verilog parser so that once a run has been performed, it can be analysed further by pointing the command line tool to the generated file.

In addition to that, a configuration file is supported to configure the different properties about the generation of the Verilog file.

Chapter 4

Implementation

This chapter describes the Verilog fuzzer implementation and the underlying analysis in seven steps, including the initial design choices, the Verilog representation, Verilog generation, simulator and synthesizer support, fuzzer configuration, test case reduction and main fuzz loop. After the initial design choices follows the description of the Verilog representation in AST, which is used throughout the library, as well as the transformation of the syntax representation into semantically correct Verilog. In the third part of this chapter two different random, deterministic Verilog generation methods are described. The main method that is used is a state-based generation method, whereas an alternative implementation uses a DAG to generate the circuit and converts it to Verilog. In a fourth step the support for simulators and synthesisers is detailed followed by a fifth point detailing the configuration and customisation of the Verilog fuzzer. Thereafter, the test case reduction is described and finally the main fuzz loop is being discussed, bringing all these modules together.

4.1 Analysis & Design Choices

This section describes the most important design choices that were made at the start of the project. At first, the reasons for choosing Haskell as a language will be discussed, followed by the rationale for the design choice of a single library.

4.1.1 Language Choice

Haskell was chosen for the library and executable that implements the requirements specified in Chapter 3 because it has a few desirable properties when dealing with a complex AST and other data structures. Haskell was chosen over other functional programming languages like OCaml or F#, because of strong familiarity and its good support for constructs such as Monads which facilitates handling these constructs.

First of all, Haskell is statically typed and purely functional, meaning that many errors are detected at compile time instead of later at runtime, thus speeding up debugging.

Common programming mistakes such as calling a function with the wrong argument types or returning the wrong type are already caught by the type system as it is compiled.

Furthermore, as a functional programming language, Haskell has algebraic data types. These help specify the structure of the AST in a much more concise and clear way compared to classes and overloading which are used in object-oriented like C++ or Java, because the algebraic data types can model the EBNF definition of the language exactly and even resembles the EBNF syntax. As a result, only syntactically valid programs can be represented by the AST. As the algebraic data type definition of the AST and the EBNF definition look similar, it is also much faster to write the AST definition and ensure that all the programs it can represent are syntactically valid. Haskell also has a large ecosystem of useful packages that can be reused and there are many libraries that can automatically generate the code necessary to manipulate types and generic tree structures. For example the `lens` package generates functions that can traverse and modify arbitrarily large trees and is hence used in VeriFuzz to manipulate the AST.

Finally, any pure function can simply be parallelised by running it on a different thread. Race conditions cannot occur, because the threads do not share any mutable state. Functions such as the Verilog generation or even the synthesis and equivalence checking step can be run in parallel, thus speeding up fuzzing runs greatly without having to worry about parallelism as these functions are designed.

4.1.2 Single Fuzzing Library

Another design choice was to write the fuzzer as a single library together with a command line tool, as opposed to using different bash scripts and makefiles like VlogHammer does. The first reason for this decision is to ensure that the whole code is compatible with many different operating systems, in order to avoid dependencies on UNIX command line tools. Using a single language has the advantage that only the language needs support for that operating system so that the binary can target it. Secondly, having a Verilog representation in the language using algebraic data types or classes means that the structure of the generated Verilog can be manipulated without having to parse it every time. Haskell libraries can be used to write functions that can manipulate the generated Verilog in complex ways. This would not be possible using scripts as the Verilog code would have to be manipulated purely as text. Generation of correct random Verilog would be much harder, and changing the Verilog in any way would also be difficult.

4.2 Verilog Representation

Verilog is implemented in the library using a tree that is formed from algebraic data types. It supports most constructs in the synthesisable subset of Verilog, apart from task definitions, function definitions and random access memory (RAM) and read-only memory (ROM) declarations.

To complement the AST, many functions have been defined that can access information from the tree or manipulate it in a various ways.

4.2.1 Verilog Grammar Subset

This section shows the EBNF for the chosen Verilog subset. The most notable features that are missing from this modified EBNF are function and task declarations, which are not generated by VeriFuzz. In addition to that, many net types such as `tri`, `tri1` or `wand` are not supported, as these are not supported by synthesisers. Furthermore, structures that are ignored by the synthesisers are also not present, such as specify blocks. Initial blocks are the only exception to this, because even though the synthesisable Verilog standard says that synthesisers should ignore it, in practice it is supported by all the synthesisers that are tested when targeting FPGAs, as these can accept initial values for variables.

A parser for the grammar shown below is also included in VeriFuzz. The lexer was adapted from a Verilog simulator written in Haskell [63], whereas the parser was written from scratch to work better with the VeriFuzz AST.

source_text ::= { **module_decl** { **module_decl** } }

module_decl ::= `module` **identifier** [**param_list**] [**port_list**] `;`
{ **module_item** } `endmodule`

module_item ::= **cont_assign** | **mod_inst** | **initial_construct**
| **always_construct** | **param_decl** `;` | **local_param_decl** `;` | **declaration**

param_list ::= `#` `(` **param_decl** { `,` **param_decl** } `)`

param_decl ::= `parameter` **identifier** `=` **const_expr** { `,` **identifier** `=`
const_expr }

local_param_decl ::= `localparam` **identifier** `=` **const_expr** { `,` **identifier** `=`
const_expr }

cont_assign ::= `assign` **lhs** = **expr** `;`

lhs ::= **identifier** | **identifier** `[` **expr** `]`
| **identifier** `[` **const_expr** `:` **const_expr** `]`
| `{` **expr** { `,` **expr** } `}`

initial_construct ::= `initial` **statement**

always_construct ::= `always` **statement**

declaration ::= [**direction**] [**net**] [**signed**] **identifier** [`=` **const_expr**] `;`

direction ::= `input` | `output` | `inout`

net ::= `reg` | `wire`

signed ::= `unsigned` | `signed`

statement ::= **time_ctrl** **empty_statement** | **event_ctrl** **empty_statement**
| `begin` **statement** { **statement** } `end`
| **lhs** `<=` **expr** `;` | **lhs** `=` **expr** `;`
| [`$`] **identifier** `(` { **expr** { **expr** } } `)`
| `if` `(` **expr** `)` **empty_statement** { `else` **empty_statement** }
| `for` `(` **lhs** `=` **expr** `;` **expr** `;` **lhs** `=` **expr** `)` **empty_statement**

event_ctrl ::= `@` `*` | `@` `(` [**edge**] `identifier` { `or` [**edge**] `identifier` } `)`

time_ctrl ::= `#` `[0-9]+`

edge ::= `posedge` | `negedge`

empty_statement ::= **statement** | `;`

const_expr ::= **number** | **identifier**
| `{` **const_expr** { `,` **const_expr** } `}`
| **unary_op** **const_expr**
| **const_expr** **binary_op** **const_expr**
| **const_expr** `?` **const_expr** `:` **const_expr**
| `"` [^"\n] `"`

expr ::= **number** | **identifier** | **identifier** `[` **expr** `]`
| **identifier** `[` **const_expr** `:` **const_expr** `]` | **identifier** `(` **expr** `)`
| `{` **expr** { `,` **expr** } `}`
| **unary_op** **expr**
| **expr** **binary_op** **expr**
| **expr** `?` **expr** `:` **expr**
| `"` [^"\n] `"`

port_list ::= `(` **identifier** { `,` **identifier** } `)`

identifier ::= `[a-zA-Z][a-zA-Z0-9]*`

number ::= `[0-9]*` [`'` `d`] `[0-9]+` | `[0-9]*` [`'` `h`] `[0-9a-fA-F]+`
| `[0-9]*` [`'` `b`] `[01]+`

4.2.2 Abstract Syntax Tree

The Verilog AST follows the Verilog specification as closely as possible. The goal is to design a tree that will only represent Verilog if and only if the Verilog is syntactically valid. One such example is the representation of a concatenation, which is defined as being a list of one or more expressions. Therefore, in the AST, the concatenation is represented as `NonEmpty Expr`, where `NonEmpty` is a list that can be checked at compile time to never be empty. This models the actual syntax of Verilog exactly and guarantees that any generated Verilog from the tree will be free of syntax errors.

However, this does not imply that Verilog program will be semantically valid, as the syntax specification for Verilog allows for semantically invalid programs. The reason for this is that semantically valid Verilog programs are not context-free and therefore, in order to simplify the parser and EBNF representation, Verilog syntax is described as a context-free superset to semantically valid programs. For the same reason, there cannot be a definition of the AST that would only allow semantically valid Verilog programs, because it requires a stronger type system allowing for conditions and functions to be written on types to capture the context sensitive grammar of compilable and synthesisable Verilog code. An example of a type system in which such a definition would be possible is with dependent types, as these allow for type definitions that depend on values.

4.2.3 Transformation to Semantically Valid Verilog

As any declaration of the defined AST is only guaranteed to be syntactically correct, transformations have to be defined which make the AST semantically valid in order for it to be accepted by all simulators and synthesisers without errors or warnings. Most of these transformations are defined to accept Verilog together with some contextual information and return the modified Verilog that is closer to being semantically valid. Threading the Verilog and contextual information through all these functions allows syntactically correct Verilog to be transformed into semantically correct Verilog.

One example is a function that finds all undeclared wires in a module and declares them properly at the start of the module. This is useful when generating completely random Verilog without first generating a circuit, as it ensures that at least the Verilog is correct. Another example is a function that accepts any module and creates a test bench specifically for that module with n number of pseudo-random hashes. The test bench instantiates the module it is supposed to test and runs the hashes through the module sequentially, recording all the results. Finally, a hash of all the recorded outputs is returned and can be compared. Once this function is written, test benches can be generated automatically for a module independent of what Verilog it receives, provided it can be represented by the AST.

Such functions simplify the random generation of Verilog, as the generator is not concerned with initialising nets or variables, or correctly defining the input and output ports of a module. Instead, this can be carried out after the Verilog has been generated using these transformation functions. However, even with these functions present, some constructs that lead to semantically invalid programs have to be prevented at generation time instead. One such example is the syntax rule defining the always block, which can be seen in the EBNF definition in Section 4.2.1.

From the EBNF definition, the two modules shown in Listing 4.1 are syntactically valid.

However, the standard also specifies that for synthesis, an always block must have one and only one event control statement. Therefore, both the modules specified in Listing 4.1 will fail synthesis and loop indefinitely in a simulator. Even in this case, only the second example compiles properly with the Icarus Verilog simulator, as it outputs an error for the `top1` module saying that the always block is missing a delay instead of simulating an infinite loop.

```verilog
module top1;
   always begin end // Empty always block resulting in Infinite loop
endmodule

module top2;
   always #10; // Infinite loop with 10 timestep delay
endmodule
```

Listing 4.1: Two Verilog examples that are syntactically correct according to the standard.

Therefore, to generate valid always blocks, the Verilog generation restricts always blocks by always generating an event control such as `@(posedge clk)` before generating any other statements. Even though event controls are valid statements, the Verilog generation will only generate one at the start of an always block, as otherwise it would result in Verilog that is not synthesisable.

4.3 Verilog Generation

Completely random Verilog generation directly from the AST would be possible. However, even though the generated random tree could be made semantically correct and therefore pass synthesis and simulation, it would probably not be deterministic. Two different generation methods, one main method and an alternative method, are therefore implemented to produce deterministic Verilog. The main method uses state-based generation similar to CSmith. It creates deterministic Verilog by following strict rules that prevent undefined behaviour from appearing in the design. The alternative method was implemented to compare against the main method used, it first creates a random DAG which is then transformed into Verilog by interpreting the DAG as a circuit. The resultant design is deterministic by construction, as the DAG ensures that there is only one output and no loop in the circuit. Evaluation of the effectiveness of the two methods can be found in Chapter 6.

An example module that was randomly generated using the state-based generation method can be seen in Listing 4.2. A random module generated by the alternative DAG method would look quite similar as the general structure is identical, however, there would not be any behavioural Verilog in the module. There can also only be one module in the graph generation, whereas the state-based method can generate and instantiate new

```verilog
module top #(parameter param0 = 5'h9e23848124)
(y, clk, wire0, wire1, wire2, wire3);
  // *** Declarations ***
  output wire  [(1'h0):(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'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 = {reg18,reg17,reg16,reg15,wire5,wire4} ;
  // *** Random module items ***
  assign wire4 = (((~wire5) ? ((((15'h9ecc51592fdeb04) ? reg17[(5'h1f):(2'h2)] :
  ↪   (reg18 ? wire2 : wire0)) ? $unsigned(((-2'ha73a956341f45c0) << reg18)) :
  ↪   wire1[(4'hb):(3'h7)]) - reg18) : reg15[(4'h9):(3'h7)]) >>>
  ↪   $unsigned($signed((reg16[(4'he):(3'h7)] ? ((wire5 && 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)]);
            reg17 <=  wire3[(1'h0):(1'h0)];
         end
       reg18 <=  $signed(({wire0} ~^ wire3));
    end
endmodule
```

Listing 4.2: Example of a module generated by VeriFuzz.

modules.

The general structure of the generated Verilog is shown in the comments in Listing 4.2. First, the module declaration contains a list of parameters that the module supports and which can be changed when the module is being instantiated. It then lists all the input and output wires which are declared at the start of the module. All modules will always be generated with a single output `y`, because it will contain all the internal variables and nets, which means that no other output to the module is needed. All the nets are declared and all variables are initialised to 0, so that they are not undefined at timestep 0. The output `y` is then assigned to all of the inputs concatenated together and as a result any discrepancies in any of the internal variables or nets will be detected. There is also an option to perform a reduction "xor" on the concatenation to restrict the output of the module to only one bit. This still allows for the detection of a bug if any of the internal variables or nets fail to be assigned the right value, because the one bit has to be correct for all the possible inputs to the module, which is unlikely to happen by chance. To make the bit depend on all the internal state of the module, it is defined as being the reduction of the concatenation of all the variables and wires using the "xor" operator. This reduction operation is defined with a unary '^' in Verilog and works by applying the "xor" operator in between every binary digit of a literal

$$\wedge\texttt{4'b1011} = 1 \oplus 0 \oplus 1 \oplus 1 = 1.$$

It can therefore reduce the size of any literal to 1 bit. The "xor" operator was chosen because it requires the evaluation of all the bits in the concatenation to return the correct result. Using an "and" operator would not work as in this case the synthesiser is able to set the output to a constant 0 as soon as it identifies a single 0 in the wires.

Finally comes the main body of the module, which includes all the randomly generated Verilog. The order of the module items is irrelevant, as the continuous assignment of the output `y` will ensure that `y` is always assigned to the current value of all the internal registers and nets. The body contains the randomly generated Verilog which assigns random expressions to the wires and variables declared previously. If an always block is produced, it may contain random nonblocking assignments, conditional statements and for loops which complicates the logic further.

4.3.1 Blocking and Nonblocking Assignment

Currently, VeriFuzz only generates nonblocking assignment, and therefore does not generate combinational circuits using always blocks which use blocking assignment. Combinational circuits are instead only generated using continuous assignment. The reason for this is that using nonblocking assignment together with blocking assignment, or just using blocking assignment in clocked always blocks, has many pitfalls that can lead to non-deterministic behaviour in synthesisers or simulators.

Listing 4.3a [64] shows a design that might be generated, which contains two separate always blocks with blocking assignment that depend on each other. This will already produce non-deterministic output with simulators, as the always blocks can be reordered in any way. However, synthesis produces the output that one would probably expect, as it just connects `b` to `a` with a flip-flop and `x` to `b` with a flip-flop. As the flip-flop is simulated with an always block containing nonblocking assignment, the synthesised design will look like the module shown in Listing 4.3b. Comparing these two modules will show that they are not equivalent, as in the first the always blocks could be executed in any order, whereas that would not affect the result in the synthesised netlist. As blocking assignments has these edge cases that lead to nondeterministic behaviour, only nonblocking assignment is

```
module top(x, y, clk);
   input x, clk;
   output y;
   reg a, b;
   assign y = a;
   always @(posedge clk)
     b = x;
   always @(posedge clk)
     a = b;
endmodule
```

(a) Original blocking assignment.

```
module top(x, y, clk);
   reg a;
   reg b;
   input clk;
   input x;
   output y;
   always @(posedge clk)
     b <= x;
   always @(posedge clk)
     a <= b;
   assign y = a;
endmodule
```

(b) Synthesised nonblocking assignment (generated by Yosys).

Listing 4.3: Example of blocking assignment becoming nonblocking assignment through synthesis.

generated.

4.3.2 State-based Generation

State-based Verilog generation has to ensure that the Verilog output is deterministic and only has exactly one possible output. It uses state to keep track of the current context and incrementally builds the Verilog AST, similar to the way in which CSmith generates random C files.

CSmith creates a C file line by line and keeps a state of variables and functions that are currently in scope. At each line a statement is chosen to be generated. If, for example, a function call is chosen, CSmith can either call an existing function or call a new function that has not been created yet. If the latter is the case, CSmith will first generate the new function before continuing generating the current section. This ensures that the function being called is always present. Information about the return type and the argument types are also kept in the context, so that CSmith only passes valid arguments to the function. A similar choice is made with variables, which may be chosen from a context or created either globally or locally.

As C code is executed sequentially, it makes sense that it is also generated that way. On the other hand, Verilog, being an HDL, is designed to be completely parallel. However, to produce deterministic Verilog it still makes sense to create it one statement at a time. In order to generate deterministic and synthesisable Verilog, there are a couple of rules that can be followed which indicate that sequential generation would be suitable.

- A net has to be assigned once and only once in a module
- Assignment in sequential always should only contain previously declared and assigned variables or nets

- Modules can only be instantiated if they have already been generated

To illustrate the first point, if a net were assigned more than once in a module, there would be conflicting drivers of that net which would result in undefined behaviour. This applies to continuous assignment as well as to procedural assignment for variables, however, it may be allowed in procedural blocks if the two assignments are in different branches of an if statement. In this case, the assignments would never be active at the same time and should therefore synthesise correctly. Sequential generation with a context is therefore a good method to produce these assignments, because when a new net is needed it can either be taken from the context or created. If it is taken from the context, one can be sure that it has already been assigned once. Otherwise, if it is generated from the context, one has to also generate an assignments for it and can then add it to the context and use it.

For the second point, by restricting assignments to only contain variables and nets that were declared before hand, this avoids any logic loops that could be generated. In addition to that, it means that every variable or wire that is currently being assigned is new and will not be assigned again in the module.

Lastly, similarly to functions in C, modules can not be instantiated if they are not present in the Verilog AST. This means that if a module is to be instantiated, it can either be taken from the context, in which case it has already been created, or it has to be generated and then added to the context before it can be used. Sequential generation therefore also makes sense, as modules will only be generated if they are actually needed.

Next, the state-based generation method will be described in three steps. The first part will describe the top level module generation function, which may be recursively called to generate more modules. Next, the input port generation function is detailed, which produces all the ports for the current module being generated. The final part outlines the module item generation function, which is quite similar to the functions producing other random constructs such as expressions or statements.

Module Generation

Let $\langle a; B\rangle$ denote a pair containing a and B. If A is a set such that $A = \{\langle a; B\rangle\}$, then let $B = A(a)$. Let B and B' be sets, then $\{\langle a; B\rangle\} \cup \{\langle a; B'\rangle\} = \{\langle a; B \cup B'\rangle\}$.

Algorithm 4.1 states how a module is randomly generated. This is the entry point to the random Verilog generation, as it is called to generate the top level module. When module instantiations are created, this module generation function might be called recursively to create the new module being instantiated. There are three main inputs to the algorithm:

- n which is the module name and is always “top” for the top level module.
- A read-only configuration $\mathcal{C}$, further described in Section 4.5, containing probabilities and properties that were defined by the user.

- The current context Γ which contains a list of the modules m, a list of variables and nets v and a list of parameters p that are in scope. It also keeps track of the current module depth d, the current statement depth s and finally the global name counter g used to name modules, parameters, variables and wires.

The output to the generation function is a module *mod* and the new context Γ'.

Algorithm 4.1: Module generation.

```
Input  : $n$, $\mathcal{C}$, $\Gamma$         // Module name, configuration and origin context
Output : $\Gamma''$, $mod$                    // Modified context and generated module
1  $\Gamma' \leftarrow \{\langle v;\varnothing\rangle, \langle p;\varnothing\rangle, \langle m;\varnothing\rangle, \langle s;\mathcal{C}(s)\rangle, \langle d;\Gamma(d)-1\rangle, \langle g;\Gamma(g)\rangle\}$;
2  $I \leftarrow$ GenInputPorts($\mathcal{C}$, $\Gamma'(g)$);                  // Generate input ports
3  $P \leftarrow$ GenParameters($\mathcal{C}$, $\Gamma'(g) + |I|$);         // Generate module parameters
4  $\Gamma' \leftarrow \Gamma' \cup \{\langle v; I\rangle, \langle p; P\rangle\}$;                                  // Update context
5  $\Gamma'(g) \leftarrow \Gamma'(g) + |I| + |P|$;
6  $\Gamma', M \leftarrow$ GenModuleItems($\mathcal{C}$, $\Gamma'$);
7  $L \leftarrow \Gamma'(v) \setminus I$;                              // Variables declared in module
8  if $\mathcal{C}(combine)$ then              // If output should be combined into 1 bit
9      $y \leftarrow$ Port($Output$, $Wire$, $Unsigned$, $1$, "$y$");
10     $M \leftarrow M \cup$ ContAssign($y$, UnaryXor(Concat($\mathcal{C}$, $L$)));
11 else                                 // Otherwise everything is concatenated
12     $y \leftarrow$ Port($Output$, $Wire$, $Unsigned$, SizeCount($L$), "$y$");
13     $M \leftarrow M \cup$ ContAssign($y$, Concat($\mathcal{C}$, $L$));
14 $clk \leftarrow$ Port($Input$, $Wire$, $Unsigned$, $1$, "$clk$");
15 $mod \leftarrow$ Correct(Module($n$, $P$, $I \cup \{y, clk\}$, $M$));
16 $\Gamma'' \leftarrow \Gamma$;                                        // Context is recovered
17 $\Gamma''(m) \leftarrow \Gamma''(m) \cup \Gamma'(m) \cup \{mod\}$;            // Available modules are added
```

On line 1, the new context Γ' is initialised using the configuration and current context Γ that was passed as an input to the function. All the sets containing symbols that were in scope are set to be empty, disregarding any values that were in the context. This is necessary because the current module will not have access to any of the nets or variables that were declared previously, as those are local to the module they were declared in. The same applies to any parameters that were in the context previously. The list of modules is also reset, even though these are global and would technically still be accessible. However, in order to avoid any module instantiation cycles, the current module being generated should not be allowed to instantiate already created modules, as these have a chance of instantiating the current module.

The next two lines of the algorithm generate the input ports and parameters of the module. These will return a list of random ports and parameters with random properties such as size or if they are signed or not. They therefore only need a configuration and the global name counter as inputs. The former is used to determine the size of the set, as this can be determined in the configuration. The latter is used to name the parameters and nets that are created so that the names are distinct. A single counter is used for all the possible names that are given in the Verilog source, therefore it has to be incremented by the number of inputs created before being passed to `GenParameters`. After these sets have been created, the context is updated with the corresponding changes so that all the variables and parameters are in scope. The global counter is also updated to reflect the added wires, so that none of the wire names will ever collide. An example of the context after the execution of algorithm until line 5 is

$$\Gamma = \{\langle v; \{w0, w1\}\rangle, \langle p; \{p2\}\rangle, \langle m; \varnothing\rangle, \langle s; 5\rangle, \langle d; 5\rangle, \langle g; 3\rangle\}\,.$$

The global counter is correctly set to 3 as there are two input wires, $w0$ and $w1$, where these are short for a random port of type *Wire* named “wire0” and “wire1” respectively and a parameter $p2$ which is an abbreviation for a random parameter named “param2”.

On line 6, the main module items are generated, which will make up the body of the module. This function takes in the configuration and the context, and returns a set containing different possible module items, such as continuous assignments of nets or always blocks. It returns an updated context, as new variables or modules may have been created during the execution of the function. The module items I are also returned and will be added to the module. At this point, the context might contain the following elements

$$\Gamma = \{\langle v; \{w0, w1, w3, r4, w5\}\rangle, \langle m; \{m6\}\rangle, \langle s; 5\rangle, \langle d; 5\rangle, \langle g; 7\rangle\}\,.$$

After calling the `GenModuleItems` function, the global name counter has now increased to 7 and there are three new variables or nets in the context. These are $w3$, $r4$ and $w5$, where $r4$ is a random reg named “reg4”. An extra module has also been added to the context meaning in the module items there is a module instantiation of a new random module named “module6”. The statement depth s and the module depth d have been reset correctly to the values they had before entering the function.

Finally, the actual module has to be created from the different sets that were generated. First, the output to the module has to be created from all the internal variables and nets that were declared. These can all be extracted by subtracting the inputs to the module from the context, after the context has been updated from the `GenModuleItems` function call, which is assigned to L. The output of the module can then be created, which combines all the internal variables and nets that were used and declared in the module. This can be done in two different ways depending on the configuration, either all the nets and variables are concatenated, or they are additionally reduced to 1 bit using the xor reduction operator.

If the intention is to combine the output into 1 bit, an output port with size 1 bit is created. The correct continuous assignment is produced which reduces the concatenation to 1 bit and is added to all the other module items that were already generated. Instead, if the output of the module should not be combined, then the port y is then created with the total size of all the wires and nets in L. The continuous assignment without the reduction is added to the module items instead. The module also requires a clock which sequential always blocks assume is named "clk" which is created on line 14. At the very end of the algorithm, the module is created using the `Module` function by passing the name n, the parameters P, the input ports with the created ports and finally all the module items. The `Correct` function then ensure that the module is semantically valid by declaring all the wires in the context correctly and adding that to the start of the module item.

At the very end of the function on lines 16 and 17, the context is recovered and returned, so that only the correct variables and nets are in scope, but not the ones that were declared in this module. This is necessary because this function might have been called recursively from a different module and therefore needs to recover the context of the original module. However, any modules that were generated in this module, in addition to the module itself, can safely be added to the context so that there is a chance they are instantiated again. This guards against recursive definitions of modules because they can only instantiate modules that are defined further down, defined by the module depth d.

Port Generation

Algorithm 4.2 describes the body of the function `GenInputPorts`. The purpose of this function is to generate the input ports to the module that is currently being produced. It takes as an input the configuration $\mathcal{C}$ and the name counter g. The output will be a set of ports I that are will be added to the ports of the module.

Algorithm 4.2: Input port generation for module

```
   Input  : $\mathcal{C}$, $g$                    // Configuration and name counter
   Output : $I$                                   // Input ports
1  begin
2      $I \leftarrow \varnothing$;
3      for $i \in [1, \texttt{Random}([1, \log \mathcal{C}(size)])]$ do
4          $isSigned \leftarrow \texttt{Random}(\{Signed, Unsigned\})$;
5          $size \leftarrow \texttt{Random}([1, \mathcal{C}(size)])$;
6          $I \leftarrow I \cup \texttt{Port}(Input,\ Wire,\ isSigned,\ size,\ \text{``wire''} \texttt{ ++ } (g + i - 1))$;
```

The algorithm starts by initialising I to be empty and then enters the loop, which runs for a random number of iterations. The configuration defines the number of iterations that the loop should run for using the general size parameter, which is scaled by log so that the number of input states does not scale exponentially with the size. The algorithm then proceeds by randomly choosing if the wire is signed or unsigned and by choosing the size of the port which scales linearly with the configuration size. Finally, the port is added to the set of input ports which are returned when the loop terminates. The name of the wire is set using the current loop iteration i and the global name counter g.

Module Item Generation

Algorithm 4.3 describes the `GenModuleItems` function that is used to generate a random set of module items for the current module. Many other generation functions follow the same structure in order to choose constructs using a frequency distribution that is defined in the configuration $\mathcal{C}$. The inputs to this algorithm are the configuration $\mathcal{C}$ and the context Γ. The updated context Γ' is returned together with the set of module items that are created.

Algorithm 4.3: Module item generation.

```
   Input  : $\mathcal{C}$, $\Gamma$
   Output : $\Gamma'$, $M$
1  $M \leftarrow \varnothing$, $\Gamma' \leftarrow \Gamma$;
2  if $\Gamma(d) > 0$ then $instanceProb \leftarrow \mathcal{C}(instance\_prob)$; // Check if module limit
     is reached
3  else $instanceProb \leftarrow 0$;                    // If yes, do not generate module
4  for $i \in [1, \texttt{Random}([1, \mathcal{C}(size)])]$ do
5      $\Gamma'$, $m \leftarrow$ choose do              // Choose from frequency distribution
6          freq $\mathcal{C}(always)$ do return GenAlways($\mathcal{C}$, $\Gamma'$);
7          freq $\mathcal{C}(cont\_assign)$ do return GenContAssign($\mathcal{C}$, $\Gamma'$);
8          freq $\mathcal{C}(param)$ do return GenParam($\mathcal{C}$, $\Gamma'$);
9          freq $instanceProb$ do
10             $mod \leftarrow$ choose do
11                 freq $\mathcal{C}(new\_mod)$ do return GenModule("module" ++ $\Gamma'(g)$, $\mathcal{C}$,
                     $\Gamma'$);
12                 freq $\mathcal{C}(existing\_mod)$ do
13                     if $\Gamma'(m) \neq \varnothing$ then return Random($\Gamma'(m)$);
14                     return GenModule("module" ++ $\Gamma'(g)$, $\mathcal{C}$, $\Gamma'$);
15     $M \leftarrow M \cup \{m\}$;
```

Like Algorithm 4.2, the `GenModuleItem` algorithm first initialises the set of module items to be empty, and also copies over the input context into the new context. It then checks if new modules need to be be generated by confirming that the depth is larger than 0 and assigning the correct probability to *instanceProb* if that is the case, otherwise no new module instance is generated by setting the frequency to 0. The algorithm then enters in the for loop which iterates a random number of times and is limited again by $\mathcal{C}(size)$.

In the loop, the module item is chosen from a frequency distribution using the **choose** construct. Inside that construct, each **freq** sets the frequency of executing that statement, which is taken from the configuration. A small example is shown in Algorithm 4.4, where x is assigned either $\{1, 2, 3\}$ from a frequency distribution. This means that $x = 1$ with a probability of $5/(5 + 7 + 3) = 5/15$, $x = 2$ with a probability of $7/15$ and $x = 3$ with a probability of $3/15$. Using such a frequency distribution to define how often a specific construct is picked rather than using a standard probability distribution has the advantage that the user does not need to verify that the numbers add up to 1, and can instead define these probabilities by defining the frequencies relative to the other constructs.

Algorithm 4.4: choose construct example.

```
1 $x \leftarrow$ choose do
2     freq 5 do return 1;
3     freq 7 do return 2;
4     freq 3 do return 3;
```

Using the **choose** construct with the frequencies taken from the configuration, each construct that can be inside of the module items has a chance of being picked. A construct can therefore be disabled if the configuration contains a probability of 0 for that construct. If, for example, an always block is chosen by the choose construct, then the `GenAlways` function is called with the configuration and the context and returns the updated context together with the always block. If a module instantiation is chosen instead, there is a further choice between generating a new module recursively with the `GenModule` function described in Algorithm 4.1, or an existing module from the context. However, if there are no existing modules then a new module will be generated anyways. After the module item has been chosen, it is added to the set of module items which are then returned.

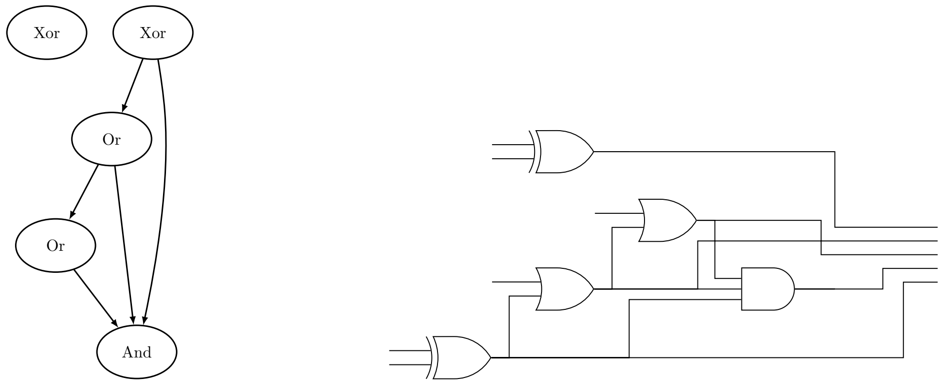
Expression and Statement Generation

Expressions and statements are generated in a similar manner with the difference that the **choose** construct will contain recursive calls and the depth of the statement or expression is tracked. The possible nodes in the structure can be split into two different types, safe nodes and unsafe nodes, where the former are not recursive and can therefore safely be generated. Unsafe nodes, on the other hand, are recursive, and may contain multiple recursive structures inside of it, which in turn may contain more unsafe nodes. If the expression or statement tree is generated naively, it would most likely never terminate as the tree would grow forever. At every node, there would be a chance to make an unsafe node, which would restart the production of an expression or statement, and potentially start the production of multiple new expressions or statements. To contain the exponential growth of the expression or statement tree, the current nesting level has to be taken into account when choosing which nodes to generate. If the level has reached the maximum depth that it was assigned, only safe nodes can be produced and the growth of the tree will be stopped.

4.3.3 Alternative DAG Generation

Random generation of Acyclic graphs is implemented by generating a list of nodes and randomly connecting a node to another node further down in the list. This ensures that there are no cycles, because it is impossible for an edge to point back to a node that appears earlier in the list. As the nodes are connected at random, it is possible that there are two identical edges in the graph. These edges are left in the circuit as they do not affect the result in any way. At the same time, random gates are assigned to each node generating a random circuit. The resulting graph is shown in Figure 4.1a.

The AST is generated from the DAG by a function that recursively traverses the DAG and generates wires for every node it encounters. It then assigns an expression to the wire which consists of all the inbound edges to that node. All these assignments are then placed in a module, where the inputs are the nodes in the graph that had no inbound edges. An extra wire for the output of the module is then created and assigned to the concatenation of all the wires that were declared. The sizes of all the wires in the module are also randomised, and the output wire is assigned the total size of all the wires combined. This conversion can be observed in Figure 4.1b.



(a) Example of a small generated DAG. *(b) The resultant digital circuit.*

Figure 4.1: Conversion from DAG to circuit in Verilog.

The wires are combined at the output to detect any discrepancies in the internals of the module, so that any change in the internal state of the module will cause the output to be different. Otherwise there might be a chance that for all inputs the outputs are equivalent, but that there was a bug in one of internal assignments that had been be optimised out. After the conversion of the circuit to an AST, it can be mutated further by applying random unary operators to random wires, without affecting the deterministic output of the circuit.

4.4 Simulator & Synthesiser Support

All simulators and synthesisers have widely different interfaces and often require many different files to run properly. Therefore, support for each simulator and synthesiser has to be implemented individually. The goal is to provide an abstraction from these different implementations so that it becomes easier to work with different tools. Abstraction furthermore facilitates adding tools in the future, as the tools only have to implement the given interface and will automatically work in the rest of the library. There is no interface for equivalence checkers, as Yosys already provides a wrapper around the most common equivalence checkers and provides a common interface to interact with them.

This is done by providing a tool interface that both simulators and synthesisers have to implement, and a simulator specific, as well as a synthesiser specific interface. All the functions that need a simulator or synthesiser will take the interface as an input, which means that any tool that provides that interface will automatically work everywhere else.

Currently, the tool implementation only requires the implementation of a function that returns the name of the current tool being used. This is needed when logging information about that tool, but also when new directories have to be created. If in the future there are more functions that both simulators and synthesisers need to implement, they can be added to that interface. The simulator and synthesiser interfaces are described in sections 4.4.1 and 4.4.2 respectively.

4.4.1 Simulators

The simulator interface provides a function to run a simulation of a Verilog file using a set of inputs which are passed as a list of byte strings. It requires the simulator to return a hash of the output of the module being tested after each input in the list is passed to it. The simulator that is implementing this interface then has to generate a test bench with the list of byte strings as an input to the module in every clock cycle. The test bench is not passed as an argument to the function, as those will be different for each tool. For example, Verilator, which is not currently supported, will compile the Verilog design to C++ code, which is driven with a test bench that is also written in C++. This will require a very different implementation to another simulator that simulates the design using a more traditional Verilog test bench.

Icarus Verilog

Icarus Verilog is currently the only simulator that is supported, as the focus has been on fuzzing synthesisers. It implements the simulator interface by defining a function that takes in the Verilog source and a list of byte strings and generates a test bench. A small example of the test bench can be seen in Listing 4.4.

The test bench works by instantiating the module being tested, and then changing the input on every negative clock edge. At every positive clock edge, the output of the module

```
module testbench;
   reg [2:0] in;
   wire [2:0] out;
   reg clk;
   top DUT(.y(out), .clk(clk), .wire1(in));

   initial begin
      clk = 0;
      in = 2'b10;        // Set in to 2'b10 at the start
      #10 in = 2'b01;    // After a delay of 10 timesteps assign 2'b01 to in
      #20 in = 2'b11;    // After a delay of 20 timesteps assign 2'b11 to in
      $finish;           // Finish the simulation
   end

   always #5 clk = ~clk; // Invert the clock every 5 timesteps

   always @(posedge clk)
     $strobe("%b", out); // Print the output at every clock edge
endmodule // testbench
```

Listing 4.4: Example of a generated test bench for a DUT.

is returned so that it can be read and hashed. The `$strobe` is used to print the output, because it is a function that will execute at the very end of the Verilog event queue, after all the wires in the module being tested have settled.

4.4.2 Synthesisers

The synthesiser interface needs the definition of a few functions to be defined to work properly. First of all, it needs to define a function that, if given a synthesiser, will return the name of the file that the synthesises Verilog code should be output to. It then also requires a function that, given a synthesiser and a file path, will return a new synthesiser instance with the output file updated to the given file path. Finally, the most important function that has to be implemented is the function that will run the synthesis when given the tool and the Verilog it should process. The function returns either a pass, or a failure with a description of what failed.

Yosys

Yosys is primarily a synthesiser, however, it also supports formal verification and simulation. It is the main tool used by the fuzzer as it is open source and therefore can be used freely. It also has synthesis support for many different FPGA's and is as a result quite versatile.

The implementation of the main synthesis function is quite simple, as Yosys does not require any configuration files or other scripts. All the required options can be passed directly from the command line. Therefore, the function is implemented by writing the Verilog to a file and passing it to Yosys with the right command line arguments. The result is a synthesised Verilog file which is moved to the correct output.

However, Yosys is also used to perform the equivalence check between the different synthesised files. First of all, Yosys provides support for ABC [34], which is a verification tool that accepts a subset of Verilog and can perform operations like proving assertions or doing an equivalence check. In addition to that, Yosys also provides a back end that can output SMT-LIBv2 from a design. This is useful as SMT-LIBv2 is accepted by most SAT and SMT solvers, meaning that all these tools can now be used to formally verify Verilog and prove properties. In the fuzzer library, ABC is mostly used to perform formal verification, as it tends to be more efficient than an SMT solver. However, this can easily be substituted by passing an argument to the formal verification function to use a different solver like z3 [65] if it is installed. This is managed by a formal verification tool called SymbiYosys [66] that is a wrapper around Yosys and provides a simple configuration file to manage the formal verification.

The equivalence check is done by creating a top module to prove the equivalence of the two synthesised modules. An example of such a top module is shown in Listing 4.5. It defines two output wires that are fed to the two modules under test, and one input that is also fed to the two modules. At every clock edge, the equality is then checked between the two outputs. If this property is proven to be true, then the two modules are equivalent. This can be done with ABC by comparing the structures of the two modules, or with an SMT solver by performing temporal induction.

```
module top(y_1, y_2, clk, wire0);
   output [2:0] y_1;
   output [2:0] y_2;
   input        clk;
   input [2:0]  wire0;
   top_1 DUT1(.y(y_1), .clk(clk), .wire0(wire0));
   top_2 DUT2(.y(y_2), .clk(clk), .wire0(wire0));
   always @(posedge clk)
     assert (y_1 == y_2);  // prove that outputs are equivalent
endmodule
```

Listing 4.5: Example of a generated top level module for a equivalence checking of two DUT.

XST

XST is the synthesiser that comes with the ISE design suite by Xilinx. It is the predecessor of Vivado, but is still used quite a lot as it supports many different Xilinx FPGA architectures that are no longer supported by Vivado. To synthesise a file with XST, a few configuration files have to be created. First of all, it needs a project file that defines all the input files. It then requires a Tcl command file describing what to do with the input files it is given.

The output of running the XST synthesiser with those two files present is a proprietary netlist, which subsequently has to be converted to Verilog using the `netgen` command which comes with ISE. As the netlist targets Xilinx FPGAs, it will contain many module instantiations of constructs that are normally found on an FPGA such as instantiations of a LUT or a specific buffer. However, their module definitions will not be present in the synthesised Verilog file, so in order to simulate and check the equivalence of the synthesised file, these definitions have to be provided. XST bundles definitions of these modules in their simulator ISIM, so these can be copied over and just need a few modifications to be compatible with Yosys and the formal verification workflow, such as removing tri-state logic and global clocks.

Vivado

Vivado is the current synthesiser that is bundled in the Vivado Design Suite by Xilinx, and supports all the newest Xilinx FPGAs. It only needs a Tcl file describing the commands that it must execute to synthesise a Verilog file and output a synthesised Verilog netlist. However, similar to XST, the netlist will contain module instantiations that have to be provided, which can be found in the Vivado distribution.

Quartus

Quartus is the last synthesiser that is supported, however, there are quite a few limitations which mean that it cannot be fuzzed properly. Unlike the other synthesisers, it provides many different scripts that have to be used together to create the desired synthesised Verilog netlist. The steps to synthesise and output a Verilog file with Quartus are the following

- Map the input Verilog to the desired FPGA architecture, which is Cyclone V in our case.
- Fit the mapped output to a specific FPGA, which is taken to be the largest possible configuration.
- Run the EDA tool to convert the synthesised design to a Verilog file suitable for simulation.

There are quite a few limitations with Quartus. First of all, as it requires fitting on an actual FPGA before it can be converted to Verilog that can be simulated, the Verilog size is limited by the physical size of the FPGA. The problem comes from the number of I/O ports that are present, as the larges Intel FPGA can only have a maximum of 1152 I/O bits [67]. This greatly limits the size of the generated Verilog, due to the way the output of the modules is checked. As all the internal nets are connected to the output of the module, only small modules with be able to fit on the FPGA, and as a result it becomes tricky to fuzz Quartus properly.

Another limitation is that like Vivado and XST, Quartus also uses many module instantiations to modules that are not provided in the synthesised design. However, in Quartus, these modules are not provided in the distribution, as they are encrypted IP blocks which can only be read by ModelSim, the simulator that comes with Quartus. Some of these modules have been manually implemented, but there are many more complicated modules that would take too long to implement correctly. One of these modules is the main flip-flop module, which is complex as it has many different inputs that are not well documented and are used in various different ways. It is therefore difficult to properly implement the flip-flop module. For that reason, Quartus cannot be fuzzed with random behavioural Verilog in always blocks, as that would generate flip-flops which are not implemented correctly. The only way to fuzz Quartus is by generating continuous assignments.

Identity

Finally, an identity synthesiser is also implemented, which comes in useful during the fuzz runs described in Section 4.7. This synthesiser simply writes the Verilog it received given to the output file.

4.5 Configuration

This section describes how the fuzzer can be configured, as it has been designed in order to allow for full customisation of the generated Verilog and how the fuzz runs are executed. When starting a fuzz run, a configuration file can be passed to VeriFuzz on the command line which defines many different parameters used during generation and execution. There are four main sections in the configuration file, the information section, the probability section, the property section and finally the synthesiser section. These can be seen in Listing 4.6 which shows the default configuration file that can be created by VeriFuzz.

Information section Contains information about the command line tool being used, such as the hash of the commit it was compiled with and the version of the tool. The tool then verifies that these match the current configuration, and will emit a warning if they do not. This ensures that if one wants a deterministic run and is therefore passing a seed to the generation, that it will always give the same result. Different versions might change some aspects of the Verilog generation, which would affect how a seed would generate Verilog.

Probability section Provides a way to assign frequency values to each of the nodes in the AST. During the state-based generation, each node is chosen randomly based on those probabilities. This provides a simple way to drastically change the Verilog that is generated, by changing how often a construct is chosen or by not generating a construct at all.

```
[info]
  commit = "e4737c37c9dc358d56dbb7a97d68de2c93053c0c"
  version = "0.3.0.0"

[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.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

[[synthesiser]]
  description = "yosys"
  name = "yosys"
  output = "syn_yosys.v"

[[synthesiser]]
  description = "vivado"
  name = "vivado"
  output = "syn_vivado.v"
```

Listing 4.6: Default configuration file output by VeriFuzz.

Property section Changes properties of the generated Verilog code, such as the size of the output, maximum statement or module depth and sampling method of Verilog programs. This section also allows a seed to be specified, which would mean that only that particular seed will be used in the fuzz run. This is extremely useful when wanting to replay a specific failure and the output is missing.

Synthesiser section Accepts a list of synthesisers which will be fuzzed. These have to first be defined in the code and implement the required interface. They can then be configured by having a name assigned to them and the name of the output Verilog file. By each having a different name, multiple instances of the same synthesiser can be included in a fuzz run. The instances might differ in the optimisations that are performed, or in the version of the synthesiser.

This configuration file is also printed after each run with the seed that was used. That way, the run can be rerun by simply pointing VeriFuzz to the configuration file in the run's directory.

4.6 Test-case Reduction

The reducer is an important part in evaluating any bugs that are found during fuzzing. If the equivalence check between the original design and the synthesised file fails, analysing the whole design does not give any indication why that was the case. Even though the verification step gives a counter example that shows the discrepancy, it is still hard to find the statement or expression that is not being evaluated correctly. It is important to localise the nature of the bug, because that makes it possible to compare to other bugs that have already been found and analyse if it is unique. In addition to that, submitting a bug report to the tool's maintainers requires a minimal, complete and verifiable example (MCVE) for the bug report to be valuable, as the maintainers will not look at a bug that first requires them to debug a randomly generated Verilog.

There are a few important rules that the reducer must follow in order to correctly reduce the Verilog code into a MCVE that still displays the same bug as in the original design. First of all, the reducer must be able to find out if the current Verilog code is interesting or not. This is done by passing the reducer a function that evaluates to a boolean specifying if the result was interesting or if the bug is not present in the code anymore. With that function, the reducer can always determine if it is on the right track or not. Furthermore, the reducer should never produce semantically incorrect code, as that would introduce undefined behaviour. This would therefore mean that the synthesiser can return any value, which is unlikely to be equivalent to the original register-transfer level (RTL) and would therefore reduce to a MCVE that shows the undefined behaviour but not the original bug that triggered the reduction.

Given a piece of Verilog that is known to fail in some way and a function that can check if any other piece of Verilog fails in that same way, the reducer works by performing a greedy depth-first search on the binary tree of possible partitions of the original Verilog code. By using a greedy algorithm, the assumption is made that there is only one bug in the test case, meaning that if a reduced version of the Verilog code turns out to be interesting, the rest of the Verilog code does not have to be checked any further. The greedy search itself is performed at four different levels of granularity, the modules level, module item level, statement level and finally expression level. At each level, the reduction algorithm will identify the smallest Verilog program that still gives an interesting test case.

By performing the depth first search at different levels of granularity, it will eliminate as much code as possible in each iteration of the reduction. It will first get rid of all the modules that do not contribute to the interesting test case, which means that in the next level it will not have to deal with as many module items as in the original code.

Algorithm 4.5: Top level reduction.

```
Input  : $S$, Check;  // Source code and check for interesting behaviour
Output : $R'$;                                          // Reduced source
1 $R \leftarrow$ Reduce(HalveModule, Check, $S$);        // Binary search on Modules
2 $R' \leftarrow \varnothing$;
3 for $module \in R$ do
4     $M \leftarrow$ Reduce(HalveModItem, Check, $\{module\}$);
5     $M \leftarrow$ Reduce(HalveStatement, Check, $M$);
6     $M \leftarrow$ Reduce(HalveExpression, Check, $M$);
7     $R' \leftarrow R' \cup M$;
```

Algorithm 4.5 shows the top level reduction algorithm, which takes a function `Check` that checks that the test case is interesting and the source S to be reduced, and returns the successfully reduced source R. The reduction is then performed in multiple steps with different levels of granularity so that the large chunks are removed before the reduction continues. The first argument to the `Reduce` function is a function that executes a strategy, which determines how the piece of code gets reduced. First, the modules are reduced, which requires the whole initial source. After that is done and the minimal amount of modules that showcase the bug have been found, the rest of the reductions will be applied. However, these only work on one module at a time, which means that all the remaining modules in the source are iterated over using the for loop and reduced one after another. First the module items are reduced, which includes always blocks and continuous assignments, then the statements and finally the expressions are reduced.

4.6.1 Reduction Algorithm

Algorithm 4.6 describes the recursive `Reduce` function that is called in Algorithm 4.5. This function has three inputs, the `Replace` function, the `Check` function and finally the source S to be reduced.

The `Replace` function is at the heart of the reducer, as it is the function that determines what paths will be explored by the reducer and how the source S will be broken down. The `Check` function checks the reduced code is still interesting by synthesising it again and if necessary, checking that it is still not equivalent to the original design. Finally, S is the Verilog source that is known to be interesting and should be reduced. First, the `Replace` function is called on the source which return various types of possible replacements. The switch statement then performs a pattern match to extract the actual reduction that was

Algorithm 4.6: Reduce the Verilog code based on interesting feature.

```
1  Function Reduce(Replace, Check, $S$)
2      switch Replace($S$) do
3          case Single $r$ do
4              if Check($r$) then return Reduce(Replace, Check, $r$);
5          case Dual $l$ $r$ do
6              if Check($l$) then return Reduce(Replace, Check, $l$);
7              if Check($r$) then return Reduce(Replace, Check, $r$);
8          case None do
9              return $S$;
10     return $S$;
```

performed. If it was a `Single` replacement, then it is checked if it is still interesting, and if that is also the case, then it is further reduced by calling the `Reduce` function recursively. If not, it exits the case statement and the original source is returned, meaning that the smallest replacement was found. In case it is a `Dual` replacement, two different replacements were found. A greedy search is performed on the two possible reductions, meaning if the first one is found to be interesting, the other is discarded. Finally, if there are no interesting reductions that were found, the original source is returned as the smallest reduction was reached.

The type of the `Replace` function is illustrated by example of the `halveModule` replace function

```
halveModule :: Verilog -> Replacement Verilog
```

where `Verilog` is the type of the source, and `Replacement` can either have value `Single r`, signifying there is only one reduced path to be explored, `Dual l r` meaning there are two different possible reductions or `None`, meaning there were no possible reductions.

To explain how the `Replace` functions work, the `halveModule` function is described in detail in Algorithm 4.7. The goal of this function is to take Verilog source as input and return a possible reduction on the code, which may result in two possible reductions, only one reduction or no reduction. The `halveModule` function works specifically at the module level to perform the reductions.

As shown previously in the type definition of the `HalveModule` function, it takes in the Verilog source S and returns a replacement R of the Verilog source. First of all, the top module in the Verilog code is extracted and assigned to T. The algorithm then enters an if statement that checks how many modules are present in the Verilog source. If only one module is present, then `None` is assigned to R and is returned, as that means only the top module is left and there are no possible replacements to be performed at the module level.

Algorithm 4.7: `HalveModule` function.

```
Input  : $S$                                   // Verilog source
Output : $R$                // Possible replacement for source
1  $T \leftarrow$ FindTopModule($S$);
2  if $|S| \leq 1$ then $R \leftarrow$ None;
3  else if $|S| = 2$ then
4      $S' \leftarrow$ RemoveAllInstOf($S \setminus T$, $T$);
5      $R \leftarrow$ Single($S'$);
6  else
7      $S_1 \leftarrow \varnothing$, $S_2 \leftarrow \varnothing$;
8      for $m \in S \setminus T$ do
9          if $|S_1| < |S_2|$ then $S_1 \leftarrow S_1 \cup \{m\}$;
10         else $S_2 \leftarrow S_2 \cup \{m\}$;
11     $S_1' \leftarrow$ RemoveAllInstOf($S_2$, $S_1 \cup T$);
12     $S_2' \leftarrow$ RemoveAllInstOf($S_1$, $S_2 \cup T$);
13     $R \leftarrow$ Dual($S_1'$, $S_2'$);
```

Otherwise, if there are two modules present then a single reduction on the code is possible. As the top module is always present in the Verilog code, T will be the `Single` replacement that is returned. However, to keep the code semantically valid, all the instantiations of the module that was removed have to be taken out of the module items in T. This is done by first selecting the removed module using $S \setminus T$, and calling the function `RemoveALlInstOf`, which removes all the instantiations present in any of the modules in the second argument if they refer to any of the modules in the first argument. S' therefore only contains T, but with all the traces of the other module removed and it is assigned to R as a `Single` replacement.

The last case occurs when there are more than two modules in the Verilog source. This signifies that the Verilog will have to be split into two. However, the top module T should still be present in both modules. Therefore, a for loop is entered that iterates over all the modules apart from the top module, and splits them into sets S_1 and S_2 of approximately equal sizes. Thereafter, all the instantiations in all the modules have to be removed if the modules are no longer present in the Verilog source. Again, this is carried out by the `RemoveAllInstOf` function, which first removes all the instances in $S_1 \cup T$ that refer to any modules in S_2, as those were the modules that were removed. The same process applies to S_2, so that S_1' and S_2' are semantically valid again. These are then returned as a `Dual` replacement.

4.7 Fuzz Loop

The fuzz loop is the main logic in the fuzzer that ties all the different modules together. It is responsible for managing the current fuzz runs by organising where each run should take place so that these do not interfere in any way. An important feature of the fuzz loop is that it should be isolated from the main environment and thread that the fuzzer is running in. Isolation guarantees that that multiple fuzz loops can be launched in parallel without interfering with each other. They can therefore safely change directory or change environment variables that do not affect the other runs or the main program.

Haskell uses a construct called the `IO` monad to interact with impure computations. However, the `IO` monad is quite close to the system, and as VeriFuzz interacts with quite a few programs through the command line, it would be extremely difficult to make the run on different platforms. To solve this problem, the `Sh` monad is used instead of the `IO` monad, because it provides a way to describe a computation using a local environment that is copied from the main program environment when it is created. All the necessary computations can then be described in terms of the `Sh` monad and one can be certain that the global environment will not be modified. This allows for a cross-platform way of describing the computations that need to be performed using the simulators and synthesisers. There is an additional valuable advantage to the copied environment because it makes it safe to change directories and change the environment even when only operating on one thread, knowing that none of the other parts of the program are affected. As a result, multiple fuzz runs that are described in terms of this monad can be run in parallel safely, even on the same thread if one wanted to.

The fuzz loop is shown in Algorithm 4.8. It accepts a set of synthesisers to fuzz S, a Verilog generator G and a configuration file $\mathcal{C}$. This means that the way in which the Verilog is generated can easily be changed.

The main fuzz loop is contained in a for loop that will run until the specified amount of iterations has been reached. In each iteration, a new Verilog source s is randomly generated using the `GenSample` function. This function either takes the first random sample that is generated, but it can also pick the generated Verilog sources from a distribution, so that they have a more predictable size. Once the Verilog is generated, another for loop is entered which iterates over all the synthesisers S_n that are to be tested. The source is passed to each synthesiser, and if it finished correctly, it is checked for equivalence against the original Verilog. If that succeeds as well, the netlist is run through all the simulators, and the simulation results are stored in R_s.

Finally the test case is reduced for all the failures that were found in the synthesis stage, simulation stage or equivalence checking stage. A report is generated for the current run and is added to the set of reports that will be returned.

Algorithm 4.8: Describes the main fuzz loop.

```
   Input  : $\mathcal{C}$, $S_n$, $S_m$, $G$    // Config, synthesisers, simulators, generator
   Output : $R$                                             // Set of reports
1  $R \leftarrow \varnothing$;
2  foreach $[1, \mathcal{C}(max\_iter)]$ do
3      $s \leftarrow$ GenSample($\mathcal{C}$, $G$);
4      for $syn \in S_n$ do
5          $syn\_result \leftarrow$ Synthesis($\mathcal{C}$, $s$, $syn$);
6          $equiv\_result \leftarrow \varnothing$, $r \leftarrow \varnothing$;
7          if Passed($syn\_result$) then
8              $equiv\_result \leftarrow$ Equivalence($\mathcal{C}$, $s$, $syn\_result$);
9              if Passed($equiv\_result$) then
10                 $R_s \leftarrow \varnothing$;
11                 for $sim \in S_m$ do
12                     $R_s \leftarrow R_s \cup$ Simulation($\mathcal{C}$, $sim$, $syn\_result$);
13         $red \leftarrow \varnothing$;
14         for $f \in$ Failed($syn\_result$) $\cup$ Failed($equiv\_result$) $\cup$ Failed($R_s$) do
15             $red \leftarrow red \cup$ Reduce($\mathcal{C}$, $s$, $f$);
16         $R \leftarrow R \cup \{$GenReport($\mathcal{C}$, $s$, $red$, $syn\_result$, $equiv\_result$, $R_s$)$\}$;
```

Chapter 5

Results

This chapter contains the results that were obtained from experiments performed with VeriFuzz to compare its effectiveness against existing tools and find as many bugs in synthesisers as possible. Five experiments were conducted in total, which explore different properties of the generated Verilog and compare different methods of generating Verilog to find the most efficient way of fuzzing synthesisers. The synthesisers that were fuzzed in these experiments were Yosys, Vivado and XST. However, testing of XST was stopped after the main fuzzing run, because Xilinx has said that it does not support XST anymore and will not fix any new bugs that are found in the tool. For that reason, no bugs found in XST were reported either. Icarus Verilog was also tested, however, no bugs were found in the simulator, which means that the results will only display bugs found in synthesisers.

At first, VeriFuzz was run continuously over the course of a month on all the supported tools to find as many bugs as possible and report them to the tool vendors. The final results are shown in Table 5.1, which shows the total number of bugs that were found and how many of those were unique and had not been discovered before. The unique bugs found refers to the bugs that could not be reduced to each other, and are therefore suspected to be caused by different bugs in the synthesiser.

Tool	**Synthesis bugs**		**Crashes**		**Total**	
	All	**Unique**	**All**	**Unique**	**All**	**Unique**
Yosys	4	2	1	1	5	3
Vivado	1136	9	457	5	1596	14
XST	539	4	0	0	539	4
Quartus*	0	0	0	0	0	0
Total	1679	15	458	6	2137	21

Table 5.1: Summary of number of bugs found in each tool.
**Quartus could not be tested properly so no bugs were found in the tool.*

Table 5.2 shows all the bug in Yosys that were reported, and also gives their current status. The three bugs that were found were fixed on the same day that they were submitted. Table 5.3 shows all the bugs that were reported for the newest version of Vivado, Vivado 2019.1. The reason that the Vivado unique bug number is much higher than the five reported bugs shown in Table 5.3 is because it contains all the unique bugs found in versions 2016.1, 2016.2, 2017.4, 2018.2, 2018.3 and 2019.1. Only those that were still present in 2019.1 were therefore reported to Vivado.

Type	**Link**	**Confirmed**	**Fixed**
Crash	`https://github.com/YosysHQ/yosys/issues/993`	✓	✓
Bug	`https://github.com/YosysHQ/yosys/issues/997`	✓	✓
Bug	`https://github.com/YosysHQ/yosys/issues/1047`	✓	✓

Table 5.2: Yosys bug reports.

Type	**Link**	**Confirmed**	**Fixed**
Crash	`https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Verilog-If-statement-nesting-crash/td-p/981787`	✓	✗
Crash	`https://forums.xilinx.com/t5/Synthesis/Vivado-2018-3-synthesis-crash/td-p/981136`	✓	✗
Bug	`https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Unsigned-bit-extension-in-if-statement/td-p/981789`	✓	✗
Bug	`https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Bit-selection-synthesis-mismatch/td-p/982419`	✓	✗
Bug	`https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Signed-with-shift-in-condition-synthesis-mistmatch/td-p/982518`	✓	✗

Table 5.3: Vivado 2019.1 bug reports.

Secondly, the efficiency of running Verilog programs with different sizes is explored, to determine the optimal program size is in terms of bugs found in a specific time frame.

Thirdly, the stability of different synthesisers are compared over different versions, to examine if there are any regressions or if the quality of the synthesisers increases with every release. This is done by comparing the stability of different released versions of Yosys and Vivado.

Furthermore, swarm testing is compared to enabling all the configuration options at once, to check if the bugs found by swarm testing are in fact more diverse than if swarm testing was not used.

Finally, the test case reducer built into VeriFuzz is compared against existing test case reducers that are either language agnostic, or originally targeted C and have been rewritten to accept Verilog. In this section, the properties of Verilog are also compared to C, to explain why language agnostic reduction is not very effective with Verilog.

5.1 Bugs Found

There are two main groups of bugs that can be found when fuzzing compilers, and therefore also when fuzzing simulators and synthesisers. The first possible bug is that the simulator or synthesiser crashes or fails in an unexpected way when compiling the test-case. This could occur in form of a crash with a core dump, which is an entirely unexpected behaviour for the synthesiser, or in form of an error code which appears although the Verilog input was valid. A second potential type of bug ocurrs in compilers as a miscompilation of the code but in synthesisers as synthesis into a net list that is not equivalent to the design. The latter bug is much harder to detect in production, as it may not be possible to formally compare the generated netlist to the input.

Individual synthesiser crashes are easier to identify because the stack trace provided by the tools once it crashes prints a stack trace showing the exact function that failed. Therefore, if the crash occurs in two different functions, then there are probably two different causes and the crashes are separate incidents. On the other hand, with bugs involving incorrect synthesis, it can be nearly impossible to tell if the two bugs are different without reducing the test-cases to their minimal representation. Even automatic reduction can take hours to complete with large test-cases, and even then they will have to be reduced further manually to be able to identify them definitely. Due to the large number of synthesis bugs uncovered, it is very difficult to identify a new bug.

5.1.1 Example of Wrong Synthesis Bugs

All the tools tested with VeriFuzz contained synthesiser bugs producing a netlist that was not equivalent to the original design. Vivado and XST had the most bugs, whereas Yosys only had a few.

Yosys Bug Example

One example of a bug that was found and reported in Yosys can be seen in Listing 5.1. The Verilog code is synthesised to the Verilog netlist that can be seen in Listing 5.1. The bug occurs when there is a shift by a value that is first multiplied. Peephole optimisations are pattern matching rules that detect a small set of specific instructions, or, in this case, small pieces of Verilog code, that are then replaced by a more efficient but equivalent set of instructions. It is picked up by the peephole optimisation pass, but is transformed into code that is not equivalent. Because the multiplication `w * 3'b110` results in `3'b000` instead of `5'b11000`, due to the value being truncated. Yosys was applying the peephole optimisation with the result that was not truncated which therefore resulted in the wrong code being generated.

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

Listing 5.1: Reported Yosys bug [1].

This issue was reported and quickly fixed by not matching shifts with multiplications that could result in overflows and will therefore be truncated.

Vivado Bug Example

Listing 5.2 shows a bug that appears in Vivado. The code snippet works by getting an input to the module through `w1` and the clock input through `clk`. It then assigns an internal variable `r1` to be 0 at the very start of the simulation, and is then assigned to 1 at a positive clock edge if `w1` is equal to `-1'b1`. However, because the literal `-1'b1` is one bit wide and `w1` is two bits wide, they cannot be compared directly. The Verilog standard says that if at least one of the operands of a binary operator is unsigned and of unequal size, then the smaller one is extended with zeros until both operands are of the same size. If, on the other hand, both are signed, then the smaller one is sign extended instead. In the case of Listing 5.2, `w1` is signed but the literal `-1'b1` is not. As they are of unequal size though, `-1'b1` is extended to `-2'b01`, which is equivalent to `2'b11`. Therefore, the output of the module `y` should only be 1 if the input was `2'b11`. Around the expression in the if-statement there are braces, which indicate concatenation. However, as there is only one element inside the braces, the concatenation has no effect.

When `w1` is set to `2'b01`, the generated netlist by Vivado will output 1 for `y`, even though `2'b11` does not equal `2'b01`. The correct output for that Verilog module is therefore the default value of the register, which is `1'b0`. When synthesising with Yosys and Quartus, the correct behaviour is observed and the output of the module will only be set to 1 if the input was `2'b11`. If the concatenation in the if-statement is removed, Vivado also produces the correct code and will output the right value. Therefore, the bug seems to be a mix between the sign extension and concatenation. However, if the expression is

[1] `https://github.com/YosysHQ/yosys/issues/1047`

[2] `https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Unsigned-bit-extension-in-if-statement/m-p/981890#M31443`

```verilog
module top (y, clk, w1);
   output y;
   input clk;
   input signed [1:0] w1;
   reg r1 = 1'b0;
   assign y = r1;

   always @(posedge clk)
     if ({-1'b1 == w1})
       r1 <= 1'b1;
endmodule
```

Listing 5.2: Vivado bug in if-statement[2]*.*

assigned directly to `r1` as shown in Listing 5.3, which should give exactly the same result, Vivado outputs the right code as well, indicating that the bug also is implicated with the if-statement expression itself.

```verilog
       ...
       r1 <= {-1'b1 == w1};
       ...
```

Listing 5.3: Direct assignment of `r1`.

5.1.2 Example of Synthesiser Crashes

Synthesiser crashes are much less common than synthesis into a non-equivalent netlist. However, when passed deterministic valid Verilog, some synthesisers unexpectedly crashed. Both crashes shown below, in Yosys and in Vivado, have been reported and confirmed to be bugs of those tools.

Yosys Crash Example

The code in Listing 5.4 caused a crash in the peephole optimisation pass in Yosys because an empty array was being indexed without checking if the index is valid. In this case, the peephole optimisation was supposed to optimise multiply and shift operations such as `foo[s*W+:W]`, where the latter specifies a range starting from `s*W` and going to `s*W + W`. However, the optimisation was not confirming if the shift amount was empty before trying to index it. As the shift amount is empty in this case, Yosys would crash because it was trying to index into an vector that was empty. It was reported to the maintainers of Yosys and fixed on the same day.

[3]`https://github.com/YosysHQ/yosys/issues/993`

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

Listing 5.4: Yosys crash resulting in a core dump due to peephole optimisation pass[3].

Vivado Crash Example

Vivado was shown to crash on multiple occasions in distinct ways, which was deduced from the different stack traces. An example of a crash in Vivado is shown in Listing 5.5.

This seems to be the minimal form of the crash bug in Vivado, which happens in library called `librdi_synth` which is most likely responsible for synthesis. It is difficult to give more details on the crash that was found, as the source code for Vivado is not available. In addition to that, even the bug reports and their progress is not made public, meaning that the fix cannot be analysed. Open source synthesisers such as Yosys make it much easier to follow the process of fixing the bug and finding out what the root cause of it was.

5.2 Verilog Size Efficiency

It is interesting to analyse how different sizes of generated Verilog affect the number of bugs found in a specific amount of time, to find out which size is optimal. Yang et al. [36] found that C code of around 8192–16384 was the best in terms of efficiency for finding new bugs in C. It may be assumed that the larger the programs are that are generated, the more bugs are found, because there is a higher possibility of the generated code to contain one. However, it was found that for compilers, 8192–16384 lines was the limit where larger programs would increase compile time and run time to such a point where the increased chance of finding a bug did not make it worth it.

Synthesisers, however, take much longer than compilers to transform the behavioural hardware description into a lower level netlist. This is because they have to transform the description into equivalent and often optimised logic. In addition to that, checking if there are any bugs in the Verilog is done by a formal equivalence check between the initial design

[4]`https://forums.xilinx.com/t5/Synthesis/Vivado-2019-1-Verilog-If-statement-nesting-crash/td-p/981787`

```
module top (y, clk, wire0);
   output [1:0] y;
   input        clk;
   input  [1:0]  wire0;
   reg [2:0]    reg0 = 0, reg1 = 0;
   reg [1:0]    reg2 = 0, reg3 = 0, reg4 = 0, reg5 = 0;
   assign y = reg2;
   always
     @(posedge clk) begin
        reg0 <= 1; reg5 <= 1;
        if (reg5)
          begin
             if (reg0); else if (wire0); else
                 if ($signed(reg3[0:0]))
                   begin
                      reg3 <= reg0; reg1 <= reg3;
                   end
                 else reg1 <= reg0;
             reg2 <= reg1[0:0];
          end
     end
endmodule
```

Listing 5.5: Vivado crash minimal example[4]*.*

and the synthesised logic. Using an and-inverter graph (AIG) for equivalence checking requires the mapping of nodes from one graph onto another, which is a problem called Graph Isomorphism. As there is no known algorithm in P for this problem, it does not scale linearly with the number of wires that are in a design. Therefore, as these two steps are time consuming, it is expected that smaller Verilog code will perform better and find more bugs than larger Verilog code, even though the latter has more chances of containing a bug.

This section will compare the efficiency of 8 runs with varying code sizes using a direct output of all the internal variables and nets of the module to 8 runs where the output has been reduced to 1 bit using the xor operator. Each of these runs lasted two days and were run in parallel.

Figure 5.1 shows the synthesis time and equivalence checking time as the code size increases for both the 1 bit output and complete ouptu. All of these test-cases were run with a timeout set at 900s for both the synthesis and for the equivalence checking. It can be observed that as code size increases, synthesis times for both the 1 bit output and complete output increases exponentially. This already shows that the larger Verilog files will most likely not be very efficient at finding many bugs. However, when looking at equivalence checking, the relationship between increased code size and time is not that clear.

Looking at the runs with output of all wires, the curve is much steeper than the synthesis time at the start, but then levels off to meet the synthesis time curve at around 4,000 lines of code. There is also an anomaly at around 1,000 lines of code, as the curve

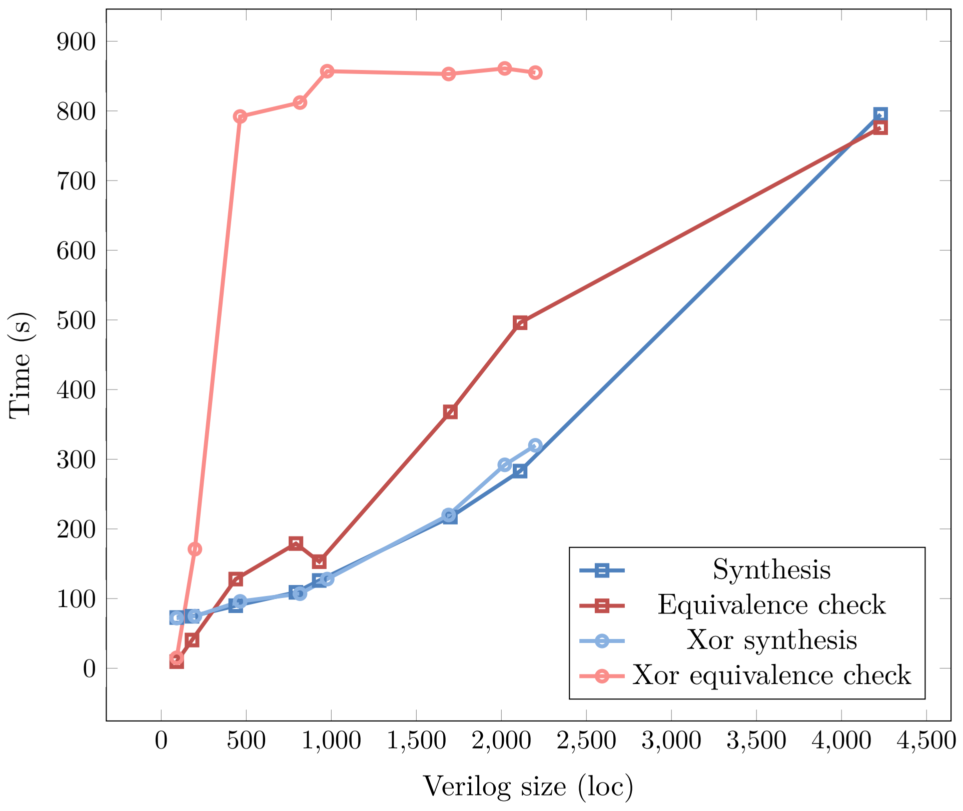


Figure 5.1: Synthesis and equivalence checking time as program size increases.

dips back down. The anomaly can be explained by the fact that the average of times is plotted and that the equivalence checking times are quite sporadic and therefore have a high variance. However, The equivalence check time levelling off is more interesting, as it seems that larger circuits are handled better than smaller circuits. The reason for this is the timeout that is set to 900s, which implies that independent of the size of the generated Verilog code, the average will converge to 900. It is therefore safe to assume that many test-cases timed out at around 4,000 lines of code, and that the trend for the equivalence checking time is also exponential.

Next, looking at the equivalence checking time for the Verilog code that reduced all the output to 1 bit, one can observe that it reaches the timeout extremely quickly, at 500 lines of generated code. The reason for this could be the large circuit that is produced by the synthesis tool to perform the reduction of all the wires in the module. As an FPGA normally only has LUTs to create gates, which often have between 4–8 spaces, the xor operation between all the bits of all the variables and nets will have to be performed in multiple stages, producing a huge circuit. The mapping of those large circuits therefore scales extremely quickly with the size of the Verilog.

5.2.1 Complete Module Output

First, the runs that were performed by outputting all the internal variables and nets are analysed using Figure 5.2.

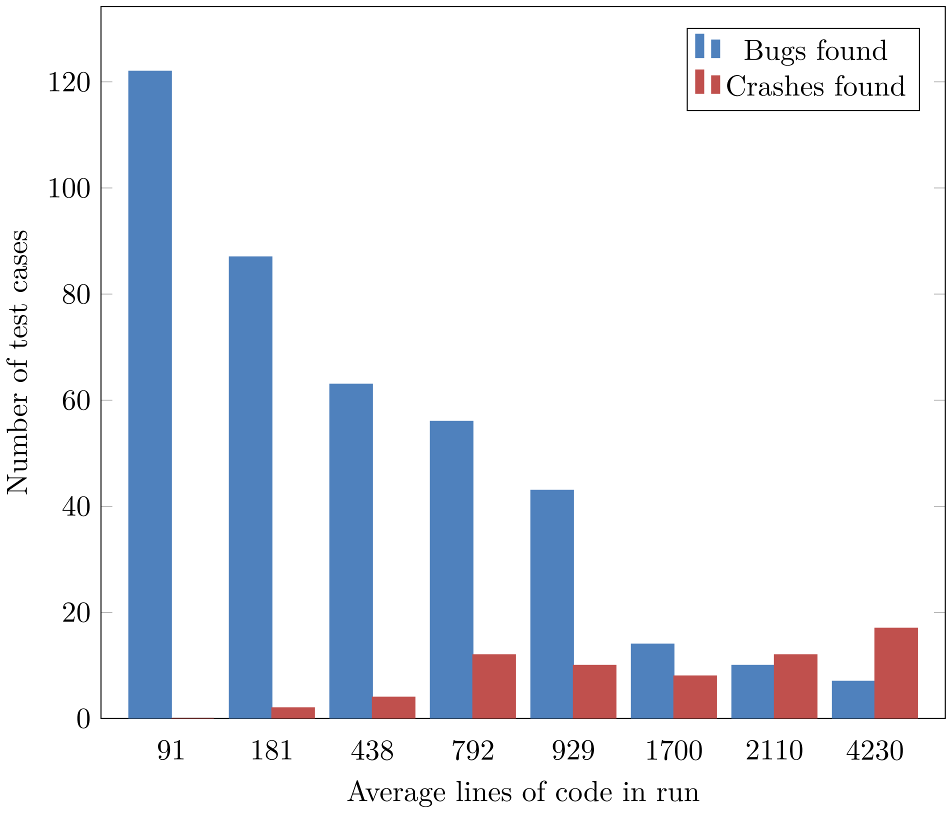


Figure 5.2: Bugs found for generated Verilog at different sizes without xor at ouptut.

Figure 5.2 shows the total number of bugs found for each separate run, showing the average lines of code in the x-axis. First of all, it can be observed that there is an exponential decrease in the number of total bugs found overall, as the size of test-cases increases. This is in line with the exponential increase of both the synthesis and equivalence checking time shown in Figure 5.1. However, as the size of the test-cases increases, so does the amount of crashes that are found in the tools, which is expected, since it is much more likely to contain a combination of Verilog that exposes an edge case in the synthesiser. As a result, it is difficult to say which size of Verilog program is the most effective at finding bugs. On the one hand, test-cases with an average size of 91 lines find the most bugs, because many more test cases can be run, but generated Verilog at that size does not find any crashes in any synthesis tools, which are also important. As a consequence one can speculate that although many bugs were uncovered with 91 lines of Verilog, these bugs are not very diverse and probalbly frequently duplicated. Therefore, there are likely many duplicate bugs that were found. With small programs choices are limited when generating Verilog, which would lead to similar modules.

Generating Verilog with 792 lines of code instead might be more advantageous, although only half the amount of bugs were found. This is because 12 crashes were found in addition to the bugs. As the program size is also larger, that means that there are many more possibilities for generating a large variety of random code, as there are many more choices that can be made.

5.2.2 Reduced Module Output

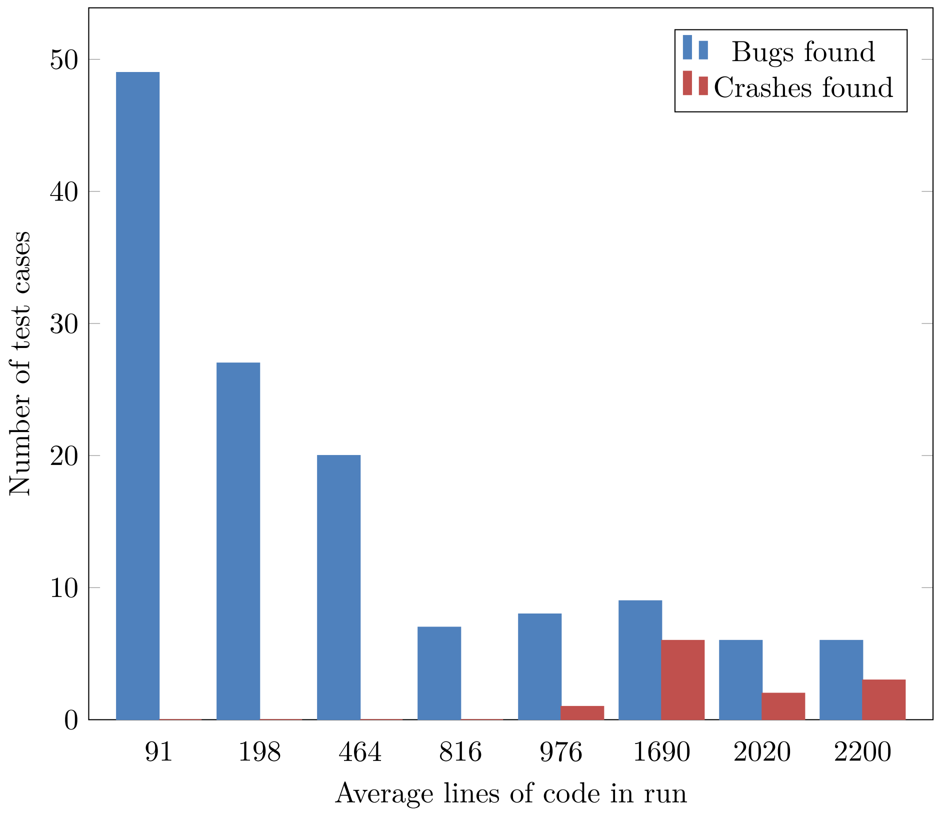


Figure 5.3: Bugs found for generated Verilog at different sizes with xor at ouptut.

The results using the xor reduction to output only 1 bit instead of all the wires combined is shown in Figure 5.3. As shown in Figure 5.1, the equivalence checking time increases dramatically as the code size increases. Already at 500 lines of code, the average equivalence checking time reaches the timeout, meaning that more than half of the equivalence checks are timing out. This can also be observed in the number of bugs found, shown in Figure 5.3, as the first three runs have a similar decrease compared to the instance when the outputs are just concatenated together. However, when it exceeds the 500 mark, the number of bugs found over time remains equal independent of the size of the code. Although the likelihood of finding a bug increases with code size, so does the probability that the test-case will timeout.

However, similar to the previous section, the number of crashes found mostly increases with code size, and no crashes are found at all at the smaller sizes. These are not affected by the timeout of the equivalence check and therefore the trend is similar.

5.3 Synthesiser Stability

Over time, there have been many different versions of Vivado. To form an idea about the development of their quality over time, each version was run with the same configuration for two days straight, and the total amount of bugs that were found in each tool was recorded. Figure 5.4 shows the total amount of bugs and crashes that occured in each tool over that period of time.

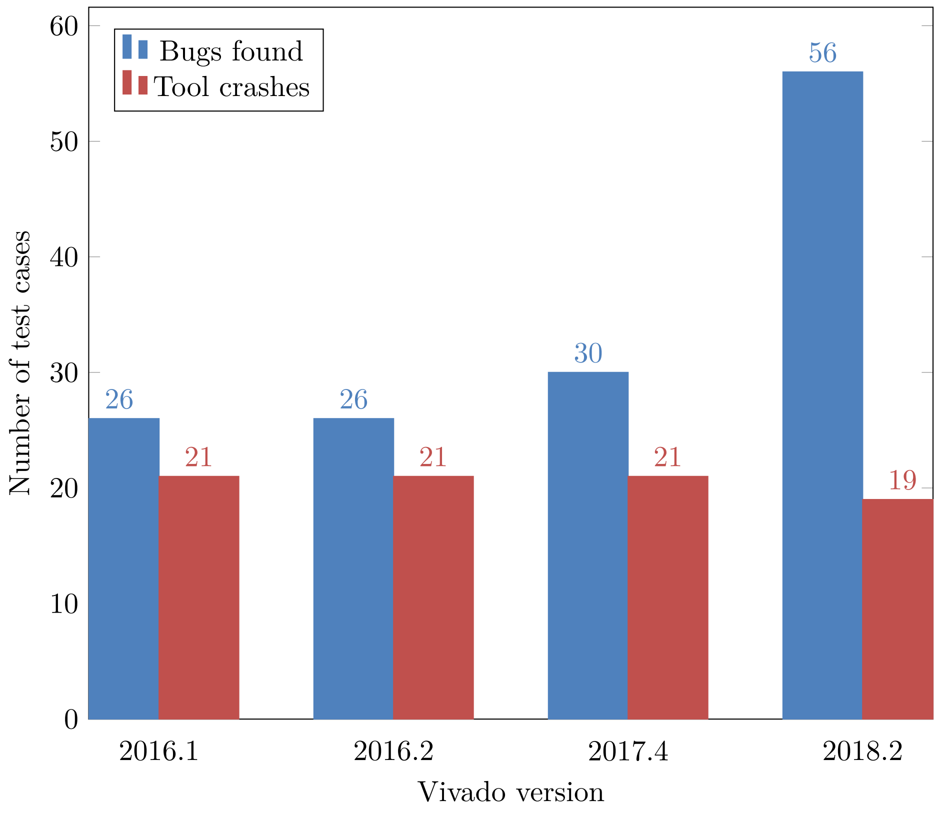


Figure 5.4: Vivado bugs in different versions

It can be observed that the number of bugs identified actually increases from year to year, however, the amount of crashes actually decreases. A number of reasons may be responsible for this. First of all, crashes are much easier to detect, as one instantly knows that the synthesis tool should not have behaved in that way. Therefore, it is much more likely that these types of bugs would have been reported earlier. Netlists that are not equivalent to the original design, on the other hand, are much harder to detect, and therefore would not be reported as often.

5.4 Swarm Testing

Swarm testing was implemented with the hopes of increasing the number of unique bugs found in the tools. Running the fuzzer constantly with the same configuration will probably find similar bugs. If there is a bug that is found quite frequently with that specific configuration, it might overshadow any other bugs that could potentially be found. Therefore, it is beneficial to randomly remove constructs from being generated.

To test the effectiveness of swarm testing against a single configuration, two runs were set up to run for 24 hours. Both were given identical configurations, meaning that they would generate programs with approximately the same size and therefore also run approximately the same amount of test cases. The difference between the two runs is that the second periodically randomises the configuration to find more unique test-cases. This randomisation was set to occur every 20 test-cases and is performed by turning off syntactical constructs with a probability of 1/2.

Testing method	**Total runs**	**Bugs found**	**Unique bugs found**
Standard method	2123	105	8
Swarm testing	2038	48	11

Table 5.4: Swarm testing tested against standard testing.

The results from the two runs can be seen in Table 5.4. It can be observed that both methods ran approximately the same amount of test cases over the 24 hours, meaning that the number of total and unique bugs found can be directly compared. This was expected, as the swarm testing only randomises the probabilities of generating syntactical constructs, and does not change the size of the generated Verilog or change any other general properties. The amount of bugs that were found using the two methods is quite different. The standard method of testing found twice as many bugs in total compared to the swarm testing methods. However, swarm testing still found more unique bugs than the standard method of generating the test-cases.

The reasons that swarm testing did not find as many bugs as standard testing, is that there were many configuration in those 2038 test-cases that did not find any bugs in the 20 runs it was used for. This is possibly the result of only using simple constructs that do not interact much with each other, or because there just are not any bugs in the tools with those constructs active. However, the number of unique bugs found is actually higher in the swarm testing method than in the standard testing method. This is because if there bugs that are found, there are only one or two that are found for each configuration. As the configuration is then changed and randomised, it is much more unlikely to find those bugs again.

5.5 Test-case Reducer Comparison

The test-case reducer is a vital part of the fuzzer, especially if one is fuzzing with large inputs. The reducer is important because it would be highly time consuming to perform this task manually. Furthermore, it is difficult to keep the test-cases deterministic, so that they show the original bug that was found. It was necessary to write a custom reducer for VeriFuzz because language agnostic reducers expect a script that tells them if a test-case is interesting in order to identify possible logical reductions and it is difficult to write a script that will check that a piece of Verilog code is deterministic. This section aims to compare the reducer build into Verifuzz to a reducer called Delta [55], which is an implementation of delta-debugging [54]. The latter was initially developed to reduce C code, however, it can be rewritten in order to reduce Verilog in a similar manner. This was done by replacing the C lexer that Delta uses to perform the splits, by a Verilog lexer.

Delta accepts a script which it runs every time it performs a reductionand which indicates whether the current test is interesting. This requires a way to check that the Verilog code is deterministic, because otherwise the reducer might introduce undefined behaviour. This would then appear as if it was a bug in the formal equivalence checker, and continue to reduce until only the undefined behaviour is present, which would not contain the original bug anymore. There have been some attempts at formulating and implementing formal executable semantics for Verilog, which would allow for a formal proof that the piece of Verilog is deterministic. For example, Meredith et al. [28] wrote a formal executable semantics for a subset of Verilog that is implemented in a tool called Maude [68]. However, only a small subset of Verilog is supported, which does not coincide with the subset that VeriFuzz produces, and it can therefore not be used. Thus, the following heuristics are used instead to make sure the design stays deterministic:

- Check that it can be parsed by VeriFuzz
- Remove uninitialised wires from all the modules
- Check that the synthesis tool does not output any warnings

The first heuristic is that VeriFuzz must be able to parse the reduced Verilog code. If that is not the case, the reduced code does not belong to the supported subset of Verilog that is responsible for the bug, and therefore is likely to contain invalid Verilog. An example is a concatenation that is reduced to an empty concatenation, which is not valid and is not accepted by the parser.

The second step is to remove all the wires that have not been initialised, so that undefined behaviour is not introduced after a reduction. This is already done by the reducer built into VeriFuzz, so it has be extended to work on parsed input as well. The reason this is important, is because if a wire is uninitialised, or undeclared, it is assumed to be `1'bz` which means that it is set to high impedance. If this wire is used in any arithmetic operations, the result is defined by the standard to be `1'bx` which is undefined. Therefore, the logic that is produced is dependent on the synthesiser and the logic will never be equivalent to the original design. The reduction tool will continue to assume that the test-case is interesting, and reduce it all the way to the undefined behaviour, which was not the original bug. Therefore, by eliminating all the uninitialised nets, this problem is avoided.

Finally, the synthesiser outputs many warnings, most of which are just informational and can be ignored, but some are important to take into account. For example, there is a warning in Vivado

```
CRITICAL WARNING: [Synth 8-6859] multi-driven net on pin
```

which displays whenever a net is being assigned twice, which would results in undefined behaviour. The way in which these are found is by disabling all the warnings that are known to be safe, such as

```
WARNING: [Synth 8-3917] design top has port y[0] driven by constant 1
```

and then checking that there are no warnings in the output at all whenever the Verilog is reduced. It is more effective to disable warnings that are known to be safe, than to only enable warnings that are known to cause undefined behaviour, because it is better to fail reducing the Verilog fully because safe warnings were ignored rather than fully reducing the Verilog due to undefined behaviour and actually losing the original bug in the process.

5.5.1 Experiment

To test the reducer that was built into VeriFuzz against the modified C test-case reducer Delta, three random failing test-cases were chosen and reduced with both tools. The time the reduction took and the final size of the reduced code are both recorded and compared against each other. Delta was configured to run reduction at all levels, which is not normally recommended as it takes much longer, but will produce the absolute smallest example test-case that it can. The script showing if a test-case is interesting was also written, which checks that the bug is still present, but also applies the heuristics to check if a test case is interesting or not. It can be seen in the Appendix in Listing A.2.

The reduced Verilog files obtained from Delta are often quite mangled and will have quite a few statements on a single line. Therefore, to compare the outputs of both methods equally, they are first parsed and pretty printed by VeriFuzz. After that is done, the lines of code give a good indication of how much the Verilog has been reduced and can be compared.

The results can be seen in Table 5.5. These show that VeriFuzz performs better than Delta in both code size and time taken. The exception is the third test-case, where VeriFuzz

Test case	Initial size	VeriFuzz reducer size (loc) / time (min)	Delta reducer size (loc) / Time (min)
1	379	32 / 5.17	61 / 375
2	618	53 / 21.13	56 / 334.5
3	2942	539 / 45.23	283 / 2040

Table 5.5: Reducer comparison results.

got stuck and could not reduce the code further. This is likely because the binary search space is not large enough to find the exact cause of the bug, however, because of this, VeriFuzz can reduce test-cases much faster than Delta, and often results in a test-case that is small enough to analyse. The large difference in time taken for reduction is especially important, as the reduction takes place in the main fuzz loop and therefore stalls any other fuzz runs. Therefore, by minimising the time taken to reduce, more bugs can be found in all the tools overall, and can be reduced to a manageable size automatically and quickly.

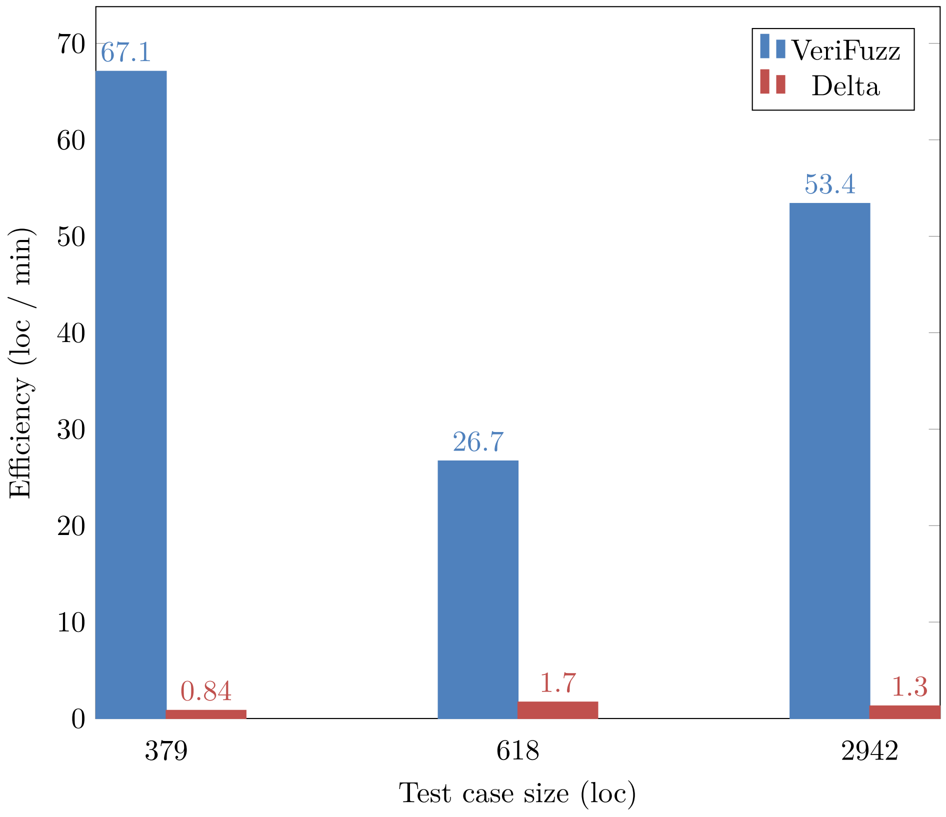


Figure 5.5: Reduction efficiency of Verilog compared to Delta.

Figure 5.5 shows the difference in efficiency between Delta and VeriFuzz. The efficiency is measured by dividing how many lines were removed from the test-case by the time taken. This shows that VeriFuzz is approximately 50 times more efficient than Delta at reducing Verilog, which greatly increases the total number of bugs that can be found.

Chapter 6

Evaluation

The aim of this project was to test various synthesisers and simulators automatically by generating random deterministic Verilog, to improve the reliability of the hardware that they generate. As there has not been much research about generating random HDLs specifically, the focus of this project was to adapt existing methods used to fuzz compilers so that they can be used to test synthesisers and simulators. To evaluate the implementation in VeriFuzz, the completion of the different objectives are compared. VeriFuzz is then compared against an existing Verilog fuzzer called VlogHammer.

6.1 Objective Completion

The main objectives that were identified to successfully write a Verilog fuzzer were the Verilog generation step and the identification step.

6.1.1 Verilog Generation

First, a DAG representing a digital circuit was used to create a deterministic circuit from which the Verilog could be generated. This method fulfilled all the criteria of being random, deterministic and correct, as the Verilog was constructed from a real circuit. However, this method was abandoned for a state-based method instead, because of the following reasons. First of all, there are many features in Verilog that are difficult to express in the DAG. For example, the ternary operator would have to be expressed using a mux, which would have to be modelled with the DAG. In addition to that, adding support for behavioural Verilog in the DAG is also not quite that straightforward. It could be implemented by detecting any loops in the circuit and inserting flip-flops in one of the edges of the loop. Even then, always blocks could only contain non-blocking assignment, as modelling for loops and conditionals in the DAG would not be possible. Secondly, the basic implementation of the DAG generation method did not find any bugs in the time it was tested for. As the Verilog is generated from an actual circuit, even though the Verilog might look random, for a synthesis tool it would be quite simple to transform it into a netlist that is equivalent to the circuit. The DAG method might not be able to exercise all the edge cases that might find bugs in the synthesiser or simulator. Finally, the DAG generation method requires the two different data structures to be maintained in parallel, and the conversion between them also has to be supported. Thus, any time a new language construct is added to the AST and should be supported by the random generation, it also has to be added to the DAG in some way and the construct also has to be handled by the function converting between the two.

Therefore, a state-based generation method was developed instead, which is able to produce behavioural Verilog in a much simpler way. As the results show, this generation method was more successful than the DAG generation method, as it found various different bugs in all the synthesisers that were tested. In addition to that, as it produces the AST directly, it does not need another representation which has to be maintained.

6.1.2 Test-case Evaluation & Reduction

Using formal equivalence checking between the netlist and the original design is a good way to identify any bugs in the synthesiser. Passing deterministic Verilog to the synthesiser means that there is only one interpretation of the design, and therefore should always be equivalent to the original design, no matter what optimisations were performed. Being able to prove if there is a bug in the synthesiser or not means that there will never be any false positives or false negatives, and that any discrepancy is therefore a bug. However, using equivalence checking also has its downsides. As was shown in Chapter 5, the equivalence checking is the bottleneck in terms of performance, as it increases with an extremely high gradient as the size of the output increases. Therefore, it is likely that with Verilog programs that are quite large, even though the likelihood of finding a bug increases, that some of the bugs are not found, as the equivalence checking times out.

A different implementation would be to use a test bench with a simulator to compare the outputs of the netlist with the outputs of the original design. This is also implemented in VeriFuzz and is used alongside the equivalence check to test the simulators at the same time, however, it was never used as a primary check for the equivalence of the netlist and design.

6.2 Comparison to VlogHammer

VlogHammer is also a synthesiser and simulator fuzzer that uses randomly generated Verilog to test these tools. However, the goals of VeriFuzz and VlogHammer are different, which explain the design choices with respect to the Verilog subset that is supported. VlogHammer focuses on testing how synthesisers and simulators handle expressions, and therefore does not support the generation of behavioural Verilog. It also contains hand-written tests alongside the auto-generated ones to test specific edge cases and also test for regressions in the tools by adding already fixed bugs to the test suite. Because VlogHammer concentrates on fuzzing all kinds of expressions, it also allows for undefined behaviour to be introduced, as those are valid expressions in Verilog. VeriFuzz, on the other hand, aims to fuzz a larger part of the language, and find bugs that have to do with multiple statements. It focuses on producing deterministic Verilog without any undefined behaviour so that formal equivalence checks can be used to find any bugs.

6.2.1 Undefined Behaviour

The main difference between the VlogHammer and VeriFuzz is that VlogHammer allows undefined behaviour, and therefore allows for non-determinism in the output of netlists. This means that two netlists might not be equivalent, but still be correct based on the design. This is unusual for a compiler fuzzer, as normally undefined behaviour is avoided at all costs. CSmith, for example, performs formal verification steps after each addition to make sure that no undefined behaviour is present in the generated C file.

Verilog is quite different to C though, and specifically allows for undefined behaviour in the standard and requires simulators to implement undefined behaviour correctly by assigning any undefined bits the valued `1'bx`. C does not have a value for undefined behaviour, and therefore allows the compiler to perform any optimisation it wants instead. This is similar to the way that undefined behaviour is handled by synthesisers, as undefined behaviour does not have a representation in hardware either. This means that netlists might not be equivalent but still adhere to the standard, and therefore results in false negatives when performing an equivalence check. The reason that VlogHammer can use undefined behaviour and still find bugs in synthesisers, is that it can detect the false negatives by running the netlists through a simulator, together with the original design, and mask out all the undefined bits that the simulator returns.

Due to this way of evaluating the output, VlogHammer can detect bugs in synthesisers and simulators that deal with undefined behaviour, and has reported many bugs that handle undefined behaviour incorrectly, which are bugs that VeriFuzz would not be able to find. However, we argue that finding bugs using purely deterministic Verilog results in more critical bugs, as even though undefined behaviour is used in real designs, it is often quite limited and can, most of the time, be refactored or replaced with Verilog that does not contain the undefined behaviour. Instead, VeriFuzz supports many behavioural Verilog constructs in always blocks that are not supported by VlogHammer, and the reported bugs to Yosys and Vivado show that there are bugs in the behavioural parts of Verilog with purely deterministic Verilog that can be found.

6.2.2 Simulator Support

VlogHammer has more simulator support, and can test Verilator, Isim, Xsim and Modelsim which VeriFuzz does not have support for yet, however, using the interface that is provided, these can quickly be added to VeriFuzz. In addition to that, Quartus can also be tested with VlogHammer, as VlogHammer does not generate code that requires flip-flops, and therefore does not need to implement it. However, Quartus can also be tested with VeriFuzz if the behavioural Verilog is turned off in the configuration.

Chapter 7

Conclusion and Further Work

Results of the experiments show clearly that the deterministic, state-based generation method implemented in VeriFuzz was the most successful and effective at finding bugs, as 2137 bugs were found in total across all the tools that were tested, and out of which 21 bugs were found to be unique. 8 of those bugs were also reported to the synthesis tool vendors. It was also found that smaller Verilog test-cases produced the most bugs, however, crashes in synthesisers were only found when the size of the test-cases was increased to 700 loc. Therefore, generating test-cases with 700 lines of code might be optimal. Swarm testing was also shown to be a valuable extension, because it found more unique bugs than the standard method of testing. Finally, it was necessary to write a custom reducer for VeriFuzz so that the reduction would only take a fraction of the time compared to using a modified instance of Delta.

Three of the reported bugs in Yosys are already fixed, and five other bugs reported to Xilinx are confirmed and will hopefully be fixed in the next release of Vivado. This shows that fuzzing using deterministic and correct Verilog is effective at finding bugs in synthesisers and can improve the overall quality of these tools.

7.1 Further Work

Even though VeriFuzz successfully found various bugs in all the synthesisers that were tested thoroughly, there are options to expand on the current project that merit being explored.

7.1.1 Extend Existing Features

The first addition that could be made is to extend the existing features that are present in VeriFuzz.

First, more Verilog constructs could be added to the Verilog subset supported by VeriFuzz, such as generate blocks, memories like ROM or RAM or Verilog attributes. In addition to that, support for combinational always blocks could also be added, which would allow for blocking assignment.

Secondly, more simulators could be supported, as currently only Icarus Verilog is used by VeriFuzz. This would allow for the proper fuzzing of the simulators as well, which likely end in finding bugs in those tools too. In addition to that, the Verilog subset could be extended to also allow features that are not synthesisable when testing simulators, which would allow for the testing of more complicated Verilog.

Finally, different equivalence checkers could also be supported, such as JasperGold [16], Conformal [69] or VC Formal [70] could also be supported, instead of relying on Yosys to invoke ABC. This would improve testing of how Yosys interprets Verilog, as one would not have to rely on Yosys parsing and converting all the netlists to a netlist format that ABC accepts. In addition to that, other formal verification tools might have better optimisations that might reduce the time taken to perform the equivalence check. This might mean that larger Verilog designs can be generated and tested, implying that the rate of bugs found could increase.

7.1.2 Undefined Behaviour

Currently, VeriFuzz is designed around the fact that the Verilog generated is deterministic, however, undefined behaviour could be introduced into the Verilog and checked in a similar way to VlogHammer. This would allow for a greater range of bugs being found. In addition to that, allowing undefined behaviour would allow for Verilog generation with much less control, and would therefore create a larger range of different test-cases that could exert interesting behaviour. However, there is the risk that introducing undefined behaviour allows the synthesisers to optimise the modules too much, which could mean that it would obscure actual bugs in the synthesisers.

7.1.3 State Machine Fuzzing

Many synthesisers perform state machine optimisations and can extract the state machine from a module. It would be interesting to see if a fuzzer could be written that fuzzes the extraction and detection of state machines, as this is not tested by VeriFuzz currently. This could be implemented by adding attributes to variables, saying that they contain the state for a state machine, even though they do not, or by constructing a random state machine that is added to the randomly produced design.

7.1.4 EMI Testing

Finally, a different way of fuzzing synthesisers and simulators might be to use EMI testing instead of differential testing. This could still benefit from formal verification, as the tool performing the equivalence check could be told to assume that one of the inputs is a constant, which makes all the alternative variations on the original program equivalent.

This would allow for the use of real designs to test the synthesisers and simulators, such as the RISC V Verilog implementation [71] by modifying it with dead code containing completely random Verilog. This could lead to more interesting bugs, as the real designs that are used are complex and could contain interesting interactions that would never be reached by generating the Verilog from scratch. A downside to this, however, could be that the synthesis and equivalence checking might take too long to find any bugs.

In addition to that, EMI testing could be combined with the random generation to produce multiple equivalent test-cases from the generated one. This would allow for the structured generation creating a deterministic design, which could then be mutated by adding dead code that is non-deterministic and may not behave properly. However, these variations should still be equivalent when given the right inputs and the netlists should therefore also be equivalent depending on those inputs.

Chapter 8

User Guide

This chapter contains a guide for the main VeriFuzz library and command line tool. First, the directory structure is described, followed by a guide detailing how to install VeriFuzz and finally a description on how to use the command line tool.

8.1 Directory Structure

The top level directory structure is the following

`app` the main code for the command-line application.

`bugs` all the bugs that have been found to date.

`data` cell implementations for modules that might be instantiated by the synthesisers.

`examples` interesting examples of different test benches and miscellaneous Verilog code.

`experiments` the configuration for the experiments that were conducted in Section 5.

`scrips` some scripts that were used during evaluation and testing.

`src` VeriFuzz source code.

`test` tests of the source code.

8.2 Installation

8.2.1 Build the Fuzzer

The fuzzer is split into an executable (in the app folder) and a library (in the src folder). To build the executable, you will need stack installed. Building directly using cabal-install is possible but not recommended and not directly supported.

To build the executable:

```
stack build
```

To run the executable:

```
stack exec verifuzz
```

To install the executable (which defaults to installing it in `/.local`):

```
stack install
```

8.2.2 Run Tests

There are two test-suites that currently test the library. One of the test-suites tests the random code generation and generation of the acyclic graph, including tests for the parser and reducer. It also contains some property based tests for this. The other test-suite uses doctest to test the examples that are in the documentation.

To run the test-suites:

```
stack test
```

8.3 Command-line Arguments

VeriFuzz provides a command-line tool called `verifuzz` which provides various commands to use the different parts of the fuzzer. The help message that is displayed with `verifuzz --help` is shown below.

```
VeriFuzz - A hardware simulator and synthesiser fuzzer.

Usage: verifuzz (fuzz | generate | parse | reduce | config)
  Fuzz different simulators and synthesisers.

Available options:
  -h,--help                Show this help text
  -v,--version             Show version information.

Available commands:
  fuzz                     Run fuzzing on the specified simulators and
                           synthesisers.
  generate                 Generate a random Verilog program.
  parse                    Parse a verilog file and output a pretty printed
                           version.
  reduce                   Reduce a Verilog file by rerunning the fuzzer on the
                           file.
  config                   Print the current configuration of the fuzzer.
```

Appendix A

Appendix

A.1 Raw Data

In the tables below, S.D. stands for standard deviation, S.T. for synthesis time, E.C. for equivalence checking time.

A.1.1 Summary of Runs Combining Output

Running with different sizes and combined output with “xor” operator.

Size	**Size (loc)**	**Var**	**S.T. (s)**	**E.C. (s)**	**Runs**	**Bugs**	**Crashes**	**Timeouts**
10	91	22	72.0	15.1	1506	49	0	5
15	198	45	74.8	171	490	27	0	63
20	464	100	96.0	792	172	20	0	122
21	816	242	107	812	157	7	0	130
26	976	249	128	857	150	8	1	136
27	1691	443	220	853	145	9	6	133
30	2018	493	292	861	140	6	2	120
31	2203	415	320	855	136	6	3	122

A.1.2 Summary of Normal Runs

Running with different sizes.

Size	Size (loc)	S.D.	S.T. (s)	E.C. (s)	Runs	Bugs	Crashes	Timeouts
10	91	21	72.8	9.50	2920	122	0	5
15	181	42	74.6	40.5	2124	87	2	63
20	438	118	89.8	128	1115	63	4	122
21	792	204	109	179	696	56	12	130
26	929	264	126	153	608	43	10	136
27	1700	399	217	368	358	14	8	133
30	2110	498	283	496	311	10	12	120
35	4230	1070	795	776	154	7	17	122

A.2 Interesting Test-case Script

Script checking if a test-case is interesting for reduction. Also checks that it is deterministic with some heuristics so that the reduction does not introduce undefined behaviour.

```bash
#!/usr/bin/env bash

last=0

verifuzz reduce -r -s identity -s vivado reduce_delta.v > output.log 2>&1

grep -E "failed" output.log >/dev/null 2>&1

if [[ $? -ne 0 ]]; then
  last=1
fi

grep -E "WARNING|ERROR" equiv/vivado.log >/dev/null 2>&1

if [[ $? -eq 0 ]]; then
  last=1
fi

rm -rf equiv

exit $last
```

Bibliography

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown", *CoRR*, vol. abs/1801.01207, 2018. arXiv: `1801.01207`. [Online]. Available: `http://arxiv.org/abs/1801.01207`.

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution", *CoRR*, vol. abs/1801.01203, 2018. arXiv: `1801.01203`. [Online]. Available: `http://arxiv.org/abs/1801.01203`.

[3] Arm, *Vulnerability of speculative processors to cache timing side-channel mechanism.* [Online]. Available: `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability` (visited on 06/18/2019).

[4] Intel, *Intel analysis of speculative execution side channels.* [Online]. Available: `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf` (visited on 06/18/2019).

[5] AMD, *AMD64 technology speculative store bypass disable.* [Online]. Available: `https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf` (visited on 06/18/2019).

[6] L. Torvalds, *Linux 4.15-rc6.* [Online]. Available: `http://lkml.iu.edu/hypermail/linux/kernel/1712.3/02898.html` (visited on 06/18/2019).

[7] Microsoft, *Windows client guidance for IT pros to protect against speculative execution side-channel vulnerabilities.* [Online]. Available: `https://support.microsoft.com/en-us/help/4073119/protect-against-speculative-execution-side-channel-vulnerabilities-in` (visited on 06/18/2019).

[8] Apple, *About speculative execution vulnerabilities in ARM-based and Intel CPUs.* [Online]. Available: `https://support.apple.com/en-us/HT208394` (visited on 06/18/2019).

[9] N. A. Simakov, M. D. Innus, M. D. Jones, J. P. White, S. M. Gallo, R. L. DeLeon, and T. R. Furlani, "Effect of meltdown and spectre patches on the performance of HPC applications", *CoRR*, vol. abs/1801.04329, 2018. arXiv: 1801.04329. [Online]. Available: http://arxiv.org/abs/1801.04329.

[10] T. R. Nicely, *Pentium fdiv flaw.* [Online]. Available: http://www.trnicely.net/pentbug/pentbug.html (visited on 06/18/2019).

[11] T. Riesgo, Y. Torroja, and E. de la Torre, "Design methodologies based on hardware description languages", *IEEE Transactions on Industrial Electronics*, vol. 46, no. 1, pp. 3–12, Feb. 1999, ISSN: 0278-0046. DOI: 10.1109/41.744370. [Online]. Available: https://doi.org/10.1109/41.744370.

[12] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The Risc-V instruction set manual, volume i: Base user-level ISA", *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[13] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, "The rocket chip generator", *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[14] K. Asanovic, D. A. Patterson, and C. Celio, "The berkeley out-of-order machine (BOOM): An industry-competitive, synthesizable, parameterized Risc-V processor", University of California at Berkeley Berkeley United States, Tech. Rep., 2015.

[15] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language", in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[16] Cadence, *JasperGold Formal Verification Platform.* [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html (visited on 01/24/2019).

[17] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Yu-Wing Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas", in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2017, pp. 1–6. DOI: 10.1145/3061639.3062244. [Online]. Available: https://doi.org/10.1145/3061639.3062244.

[18] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, Wei Song, J. Mawer, A. Cristal, and M. Luján, "An empirical evaluation of high-level synthesis languages and tools for database acceleration", in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–8. DOI: `10.1109/FPL.2014.6927484`. [Online]. Available: `https://doi.org/10.1109/FPL.2014.6927484`.

[19] Ching-Yi Wang and K. K. Parhi, "High-level DSP synthesis using concurrent transformations, scheduling, and allocation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 3, pp. 274–295, Mar. 1995, ISSN: 0278-0070. DOI: `10.1109/43.365120`. [Online]. Available: `https://doi.org/10.1109/43.365120`.

[20] Amazon, *Amazon EC2 F1 instances*. [Online]. Available: `https://aws.amazon.com/ec2/instance-types/f1/` (visited on 06/18/2019).

[21] Microsoft Research, *Project Catapult*. [Online]. Available: `https://www.microsoft.com/en-us/research/project/project-catapult/` (visited on 06/18/2019).

[22] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Springer Science & Business Media, 2006, ISBN: 978-0-387-31004-6.

[23] Y. Herklotz Grave, *VeriFuzz*. [Online]. Available: `https://github.com/ymherklotz/verifuzz` (visited on 05/27/2019).

[24] C. Wolf, *Yosys Open SYnthesis Suite*. [Online]. Available: `http://www.clifford.at/yosys/` (visited on 01/11/2019).

[25] ——, *VlogHammer*. [Online]. Available: `http://www.clifford.at/yosys/vloghammer.html` (visited on 01/11/2019).

[26] "IEEE standard for Verilog hardware description language", *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–560, 2006. DOI: `10.1109/IEEESTD.2006.99495`.

[27] "Verilog register transfer level synthesis", *IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1*, pp. 1–116, 2005. DOI: `10.1109/IEEESTD.2005.339572`. [Online]. Available: `https://doi.org/10.1109/IEEESTD.2005.339572`.

[28] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu, "A formal executable semantics of Verilog", in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, Jun. 2010, pp. 179–188. DOI: `10.1109/MEMCOD.2010.5558634`. [Online]. Available: `https://doi.org/10.1109/MEMCOD.2010.5558634`.

[29] S. Williams and M. Baxter, "Icarus Verilog: Open-source Verilog more than a year later", *Linux J.*, vol. 2002, no. 99, pp. 3–, Jul. 2002, ISSN: 1075-3583. [Online]. Available: `http://dl.acm.org/citation.cfm?id=513581.513584`.

[30] W. Snyder, P. Wasson, and D. Galbi, "Verilator", *Direct search methods: then and now*, 2007.

[31] "IEEE standard for SystemVerilog–unified hardware design, specification, and verification language", *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb. 2018. DOI: `10.1109/IEEESTD.2018.8299595`. [Online]. Available: `https://doi.org/10.1109/IEEESTD.2018.8299595`.

[32] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6", Department of Computer Science, The University of Iowa, Tech. Rep., 2017.

[33] A. Pnueli, "The temporal logic of programs", in *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, Oct. 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`. [Online]. Available: `https://doi.org/10.1109/SFCS.1977.32`.

[34] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40, ISBN: 978-3-642-14295-6.

[35] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of Unix utilities", *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: `10.1145/96267.96279`.

[36] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers", *SIGPLAN Not.*, vol. 46, no. 6, pp. 283–294, Jun. 2011, ISSN: 0362-1340. DOI: `10.1145/1993316.1993532`.

[37] X. Leroy *et al.*, "The CompCert Verified Compiler", *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[38] J. Ruderman, *Introducing JsFunFuzz*, 2007. [Online]. Available: `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz` (visited on 11/23/2018).

[39] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the Rust typechecker using CLP (T)", in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 482–493. DOI: `10.1109/ASE.2015.65`.

[40] M. Zalewski, *Announcing Crossfuzz*, Jan. 2011. [Online]. Available: `http://lcamtuf.blogspot.fr/2011/01/announcing-crossfuzz-potential-0-day-in.html` (visited on 11/23/2018).

[41] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 1492–1504, ISBN: 978-1-4503-4139-4. DOI: `10.1145/2976749.2978411`. [Online]. Available: `https://doi.org/10.1145/2976749.2978411`.

[42] M. L. Laboratory, *LL-Fuzzer*. [Online]. Available: `https://github.com/mit-ll/LL-Fuzzer` (visited on 01/24/2019).

[43] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering", *arXiv preprint arXiv:1812.00140*, 2018.

[44] B. Beizer, *Black-box Testing: Techniques for Functional Testing of Software and Systems.* New York, NY, USA: John Wiley & Sons, Inc., 1995, ISBN: 0-471-12094-4.

[45] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* New York, NY, USA: John Wiley & Sons, Inc., 2011, ISBN: 978-1-118-03196-4.

[46] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.", in *NDSS*, vol. 8, 2008, pp. 151–166.

[47] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing", *SIGPLAN Not.*, vol. 43, no. 6, pp. 206–215, Jun. 2008, ISSN: 0362-1340. DOI: `10.1145/1379022.1375607`.

[48] J. C. King, "Symbolic execution and program testing", *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: `10.1145/360248.360252`. [Online]. Available: `https://doi.org/10.1145/360248.360252`.

[49] M. Zalewski, *American fuzzy lop*, 2015. [Online]. Available: `http://lcamtuf.coredump.cx/afl/` (visited on 01/15/2019).

[50] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing", in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA: NDSS, Feb. 2017. DOI: `10.14722/ndss.2017.23404`.

[51] W. M. McKeeman, "Differential testing for software", *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[52] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing", in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012, Minneapolis, MN, USA: ACM, 2012, pp. 78–88, ISBN: 978-1-4503-1454-1. DOI: `10.1145/2338965.2336763`. [Online]. Available: `https://doi.org/10.1145/2338965.2336763`.

[53] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs", *SIGPLAN Not.*, vol. 49, no. 6, pp. 216–226, Jun. 2014, ISSN: 0362-1340. DOI: `10.1145/2666356.2594334`. [Online]. Available: `https://doi.org/10.1145/2666356.2594334`.

[54] A. Zeller, "Automated debugging: Are we close?", *Computer*, vol. 34, no. 11, pp. 26–31, Nov. 2001, ISSN: 0018-9162. DOI: `10.1109/2.963440`. [Online]. Available: `https://doi.org/10.1109/2.963440`.

[55] S. McPeak, *Delta.* [Online]. Available: `http://delta.tigris.org/` (visited on 06/11/2019).

[56] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs", in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: ACM, 2012, pp. 335–346, ISBN: 978-1-4503-1205-9. DOI: `10.1145/2254064.2254104`. [Online]. Available: `https://doi.org/10.1145/2254064.2254104`.

[57] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing", *SIGPLAN Not.*, vol. 52, no. 6, pp. 347–361, Jun. 2017, ISSN: 0362-1340. DOI: `10.1145/3140587.3062379`.

[58] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in GCC and LLVM", in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, Saarbr&#252;cken, Germany: ACM, 2016, pp. 294–305, ISBN: 978-1-4503-4390-9. DOI: `10.1145/2931037.2931074`.

[59] Intel, *Intel Quartus.* [Online]. Available: `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html` (visited on 01/14/2019).

[60] Xilinx, *XST synthesis overview.* [Online]. Available: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_using_xst_for_synthesis.htm` (visited on 01/11/2019).

[61] ——, *Vivado Design Suite.* [Online]. Available: `https://www.xilinx.com/products/design-tools/vivado.html` (visited on 01/14/2019).

[62] ——, “XST user guide”, p. 407, 2009. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf.

[63] T. Hawkins, *Verilog: Verilog preprocessor, parser, and AST.* [Online]. Available: https://hackage.haskell.org/package/verilog.

[64] C. E. Cummings *et al.*, “Nonblocking assignments in verilog synthesis, coding styles that kill!”, *SNUG (Synopsys Users Group) 2000 User Papers*, 2000.

[65] L. de Moura and N. Bjørner, “Z3: An efficient SMT solver”, in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[66] C. Wolf, *SymbiYosys.* [Online]. Available: https://github.com/YosysHQ/SymbiYosys (visited on 05/22/2019).

[67] Intel, *Cyclone v device overview.* [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf (visited on 06/18/2019).

[68] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic.* Berlin, Heidelberg: Springer-Verlag, 2007, ISBN: 978-3-540-71940-3.

[69] Xilinx, *Conformal equivalence checker.* [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/equivalence-checking/conformal-equivalence-checker.html (visited on 06/17/2019).

[70] Synopsis, *VC Formal: Next-generation formal verification.* [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html (visited on 06/17/2019).

[71] UC Berkeley Architecture Research, *vScale.* [Online]. Available: https://github.com/ucb-bar/vscale (visited on 06/17/2019).