IMPERIAL

Finding and Analysing Bugs in Verilog Consuming Tools

Author

R.A. Masih

CID: 02056535

Supervised by

Dr J. Wickerson

Quentin Corradi

Prof T. Constandinou

A Thesis submitted in fulfillment of requirements for the degree of
Master of Engineering in Electronic and Information Engineering

Department of Electrical and Electronic Engineering
Imperial College London

2025

Abstract

This project presents a novel Verilog fuzzer aimed at identifying corner-case bugs in synthesis and simulation tools such as Quartus, Yosys, Vivado, ModelSim, and Icarus Verilog. Unlike existing fuzzers, which typically focus on simple or flat Verilog designs, this tool targets underexplored features in the Verilog standard—specifically the generate construct and hierarchical naming. Both features introduce complexity during elaboration whilst hierarchical naming imposes challenges with name resolution on simulation tools. Due to differing Verilog standard implementation within tools, fuzzing these could expose inconsistencies in tool behaviour.

The fuzzer is implemented in C++ using an Abstract Syntax Tree (AST)-based framework to generate structurally correct, synthesisable, and/or simulation-ready Verilog code. The design generator produces deterministic test cases from a single random seed, enabling reproducibility. The toolchain includes wrappers for invoking synthesis and simulation tools, as well as an equivalence checker that compares simulated results with expected values derived directly from the AST.

Key deliverables include support for deeply nested generate blocks, hierarchical reference resolution (for simulation tools), and robust failure detection and logging. Tests demonstrate correctness, determinism, and portability across toolchains. By addressing gaps in existing fuzzing approaches, this project contributes a more rigorous framework for verifying Verilog-consuming tools and improving confidence in modern digital design workflows.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced.

LLMs were used to help lay out chapters where a logical flow of sections wasn't immediately apparent. In some cases, they were used to assist with rewriting sentences that were clunky, inputting the original sentence and using the generated sentence to rewrite the original sentence again.

Copyright Declaration

Contents

1 Introduction

Contents

1.1 Introduction

This project has developed a novel Verilog fuzzer with a focus on underexplored Verilog features within the fuzzer-space (notably, the generate construct and hierarchical Naming). It is designed to generate random yet valid hardware designs for the purpose of evaluating synthesis tools such as Quartus and Vivado, as well as simulation tools like ModelSim and Icarus Verilog. This project's focus on underexplored Verilog features, particularly the generate construct and hierarchical Naming, which present unique challenges to tooling, have been avoided by other fuzzers.

The objective is twofold: to establish a framework to systematically uncover corner-case bugs in synthesis and simulation tools, and to assess how well these tools conform to the Verilog standard when faced with complex but valid input designs. The fuzzer has been implemented in C++, with an emphasis on producing semantically valid code while maintaining variability and randomness.

This project addresses key questions such as:

- Can synthesis tools correctly elaborate nested generate constructs across multiple levels?

- Are simulation tools able to resolve hierarchical names accurately post-elaboration?
- How can equivalence be measured between original Verilog and tool-transformed output?

These questions are of broad relevance to FPGA vendors, hardware engineers, and EDA tool developers, as correctness issues at the tool level can lead to costly downstream design errors.

1.1.1 Generate Construct

The **generate construct** allows Verilog to replicate or selectively omit hardware *during elaboration*, i.e. before simulation or synthesis starts. It provides three familiar control forms:

- `for`-generate loops
- `if/else`-generate conditionals
- `case`-generate branches

Conceptually they behave like their software equivalents, but the result is *static* hardware, not runtime control flow. For instance, the three-bit ripple-carry adder below is described once and automatically unrolled into three full-adder instances:

```
genvar i;
for (i = 0; i < 3; i = i + 1) begin : adder_array
  full_adder fa (
    .a   (a[i]),
    .b   (b[i]),
    .cin (carry[i]),
    .sum (sum[i]),
    .cout(carry[i+1])
  );
end
```

Listing 1.1: Three-bit ripple-carry adder using a generate loop

During elaboration the tool must

1. evaluate parameters and loop bounds,
2. create unique instance names and signal slices, and

3. preserve correct connectivity across every level of nesting.

Bugs here may manifest only in deeply-nested designs, which existing fuzzers rarely explore.

1.1.2 Hierarchical Naming

A **Hierarchical Name** is a dotted path that reaches across instance boundaries, optionally anchored by `root.`[1]. An example is shown below:

```
module top;
  sub u0 ();
  initial $display("%h", $root.top.u0.ctrl.reg_q);
endmodule
```

Listing 1.2: Referencing an internal register via *root*

Synthesis tools typically disallow or restrict such cross-module references; support varies between vendors. Simulation tools (and the small amount of synthesis tools that allow Hierarchical Naming, like Vivado) resolve these during the elaboration phase of the tool, after it had been parsed. A hierarchical path must be resolved to a unique object through every generate loop, array index, and parameter value: an operation that *could* introduce corner-case bugs in deep, automatically-generated designs.

1.2 Report Structure

This report is structured as follows:

- **Chapter 2: Background** – Covers the Verilog language, reviews relevant literature, and identifies gaps in existing fuzzing tools with respect to key constructs.
- **Chapter 3: Requirements Capture** – Describes the goals and deliverables expected from the project.
- **Chapter 4: Analysis and Design** – Presents a high-level overview of the fuzzer's architecture and key design challenges encountered.

[1] Introduced to Verilog in 2005 (IEEE1364-2005)

- **Chapter 5: Implementation** – Details the construction of the tool, including relevant code generation logic and tool interface.
- **Chapter 6: Test Plan and Verification** – Describes how the fuzzer's correctness and validity were ensured via simulation and synthesis.
- **Chapter 7: Evaluation** – Evaluates the fuzzer's effectiveness across toolchains and discusses the nature of any failures or bugs found.
- **Chapter 8: Conclusions and Further Work** – Summarises the results, lessons learned, and proposes future directions for extending the tool.
- **Reflections** – Provides a personal reflection on the development process and lessons gained.

2

Background

Contents

2.1 Overview

This project aims to develop a novel Verilog fuzzer, focusing on underexplored (in the fuzzer space) and complex Verilog features, such as the generate construct and hierarchical naming, addressing gaps in current methodologies and uncovering potential features of the Verilog standard synthesis and simulation tools may struggle with.

Synthesis tools are crucial to modern digital computation, as almost all hardware systems (whether FPGAs or ASICs [1]) rely on them at some stage. As hardware complexity increases, ensuring synthesis tool reliability is critical. Translation inaccuracies can lead to costly real-world design errors since these tools are *trusted*. To validate synthesis tool correctness, equivalency checking between pre- and post-synthesis designs (or simulation to emulate this) will be implemented.

By targeting overlooked Verilog constructs and incorporating rigorous fuzzing and equivalency checking, this project aims to enhance synthesis tool evaluation.

Simulation tools, in contrast, allow designers to verify the functional correctness of Verilog designs before physical synthesis. These tools simulate the behaviour of a design in response to test stimuli, enabling debugging and validation of logic, timing, and hierarchical interactions. Unlike synthesis tools, which may ignore certain constructs or simplify designs, simulators aim to preserve the semantics of the HDL (Hardware Description Language). This makes them particularly useful for detecting functional errors or misinterpretations in Verilog. As such, simulation forms a crucial part of the equivalency checking pipeline in this project, acting as a reference model for pre- and post-synthesis verification as well as being tested as Verilog consuming tools.

2.2 The Verilog Hardware Description Language

HDLs such as Verilog are integral to modern hardware design workflows. Verilog enables designers to describe digital circuits at various abstraction levels, from behavioural models to gate-level representations[2]. It is widely used for designing, simulating, and synthesising complex hardware systems, such as FPGAs and ASICs.

A Verilog fuzzer is a tool designed to generate random, synthesisable or simulation-oriented hardware designs to stress-test synthesis and simulation tools by exploring edge cases and untested features of the language. By identifying potential bugs or inconsistencies, fuzzers contribute to improving the robustness and reliability of these tools [3]. This approach aligns with the broader concept of fuzzing, which systematically provides varied inputs to a Program Under Test (PUT) to evaluate its response and detect anomalies [4]. Unlike traditional software fuzzers that primarily focus on security vulnerabilities by feeding malformed inputs to programs, Verilog fuzzers assess synthesis and simulation tools by ensuring correct hardware behaviour and adherence to design specifications.

Synthesis tools play a critical role in hardware design, translating HDL descriptions into optimised hardware implementations [1]. Understanding how synthesis tools handle various Verilog constructs is crucial for ensuring correctness. Any inaccuracies or misinterpretations in synthesis can lead to costly design errors. Since hardware designers rely heavily on these tools, their accuracy and reliability are paramount for modern design workflows.

Simulation tools play a complementary role by enabling pre-synthesis validation of hardware

behaviour. They interpret and execute Verilog HDL directly, allowing designers to observe waveforms, verify outputs, and catch bugs before synthesis. These tools are particularly valuable for evaluating constructs that may not be fully supported in synthesis, such as hierarchical references. Since the fuzzer also targets simulation tools, equivalence checking can be carried out by comparing simulated outputs against expected values derived from the original AST. This approach ensures that discrepancies arising from tool misinterpretation are detected early in the flow.

2.2.1 History

Verilog was developed from 1983 by Prabhu Goel, Phil Moorby, Chi-Lai Huang, and Douglas Warmke at Automated Integrated Design Systems (later renamed Gateway Design Automation in 1985) [5], [6]. The language was created to simplify digital design and enable efficient simulation of hardware systems, aiming to provide a user-friendly and versatile hardware description language.

In 1990, Gateway Design Automation was acquired by Cadence Design Systems, who transferred Verilog into the public domain under Open Verilog International (OVI), now known as Accellera [7]. Verilog was then standardised as IEEE 1364-1995, commonly referred to as Verilog-95, formalising its syntax and semantics. Further revisions to the standard have introduced new features, such as the generate construct in IEEE 1364-2001 (Verilog-2001) [8].

The advent of hardware verification languages encouraged the development of Superlog by Co-Design Automation Inc. The foundations of Superlog and Vera (another HDL) were donated to Accellera, leading to the IEEE 1800-2005 standard: SystemVerilog [7]. SystemVerilog is a superset of Verilog-2005 [2], [9], with many new features and capabilities to aid design verification and modeling. Today, Verilog remains a cornerstone of hardware design flows, particularly for FPGAs and ASICs. Despite competition from newer HDLs, Verilog's simplicity and widespread tool support make it indispensable in both industry and academia.

2.3 Literature Survey

Building on the historical development of Verilog, significant research efforts have explored various aspects of synthesis and verification. This section reviews key published works and tools related to Verilog fuzzing, synthesis evaluation, and challenges associated with underexplored constructs.

2.3.1 Existing Standards

The Verilog hardware description language has evolved through several IEEE standards, each introducing new features and improvements. As discussed in Section 2.2.1, Verilog's hierarchical naming and the generate construct, introduced in IEEE 1364-1995 and IEEE 1364-2001 respectively, may pose significant challenges for synthesis tools. These constructs allow for complex, modular designs but require synthesis tools to correctly interpret and implement them. [2], [5], [8]

Despite standardisation, different synthesis tools often exhibit variations in how they handle these constructs. This inconsistency can lead to synthesis discrepancies, highlighting the need for rigorous testing and validation methodologies, such as fuzzing. Understanding the challenges introduced by these constructs provides a foundation for exploring the gaps in current fuzzing approaches.

2.3.2 Existing Verilog Fuzzing and Testing Approaches

VeriSmith

VeriSmith [3], [10] is a tool developed by Yann Herklotz, designed to automatically identify and analyse bugs in FPGA synthesis tools through random generation of correct Verilog designs. It employs a two-step process: first, it generates deterministic Verilog designs and synthesises them using tools such as Quartus, Vivado, and Yosys; then, it performs equivalence checking to compare the synthesised netlist to the original design.

VeriSmith generates valid and deterministic Verilog, ensuring accurate equivalency checks and minimising false positives. It focuses on a synthesisable subset of Verilog 2005, producing test cases that leverage various constructs, including: continuous assignments, always blocks, local parameter declarations, module instantiations, and wire and variable declarations.

VeriSmith employs formal verification techniques using tools such as the ABC verification system and SMT solvers to determine the logical equivalence of the generated and synthesised designs. Its test case reduction feature minimises failing test cases to simplify debugging by synthesis tool vendors.

The tool was extensively tested across multiple synthesis tools, including Yosys, Vivado, XST, and Quartus Prime, accumulating around 18,000 CPU hours. VeriSmith discovered several syn-

thesis tool bugs, including:

- **Incorrect synthesis output:** Vivado and Yosys were found to misinterpret Verilog constructs, leading to functional mismatches.
- **Crashes:** Both Yosys and Vivado exhibited crashes when presented with valid Verilog designs, highlighting robustness issues.

Whilst VeriSmith successfully reported eleven bugs, at least six of which have been fixed, it deliberately avoided the use of undefined values to ensure more reliable equivalence checking. However, this design choice may have limited its ability to detect certain synthesis issues.

Despite its successes, VeriSmith does not incorporate hierarchical naming or the generate construct, which are critical for evaluating complex Verilog designs. This presents an opportunity for this project's fuzzer to expand on VeriSmith's capabilities by introducing more advanced Verilog features that push synthesis tools to their limits.

VlogHammer

VlogHammer [11] is a Verilog fuzzer developed by Claire Wolf, designed to test major commercial synthesis tools such as Quartus, Vivado, and Yosys. It has discovered and reported around 75 bugs to tool vendors. Unlike other fuzzers, VlogHammer generates non-deterministic Verilog code, which necessitates an additional simulation step to filter out false positives. This approach allows VlogHammer to identify discrepancies between pre- and post-synthesis designs by applying differential testing techniques.

One key limitation of VlogHammer is its lack of support for behavioural Verilog constructs such as always blocks. It primarily focuses on synthesizable constructs and does not generate multi-module designs, which may limit its ability to uncover certain classes of synthesis bugs. Additionally, it does not perform test case reduction automatically; instead, it relies on generating smaller Verilog modules that require manual inspection when synthesis failures occurse limitations, VlogHammer serves as a crucial tool in the hardware verification space. It is accessible through open-source platforms such as GitHub, and its test cases continue to contribute to improving synthesis tool reliability. However, for this project, the focus will be on enhancing the exploration of overlooked Verilog features, such as hierarchical naming and the generate construct, areas where VlogHammer's current implementation falls short.

Fuzzing Hardware: Faith or Reality?

Mousavinezhad et al. [12] discuss the challenges and realities of applying software fuzzing techniques to hardware verification. It is important to note that this paper focuses on fuzzing a particular hardware design (the inputs supplied to it to confirm correctness of the design), whilst this project focuses on fuzzing synthesis/simulation tools by generating multiple hardware designs and confirming that the post-synthesis/simulated Verilog matches the behaviour of the original designs.

The paper highlights that while fuzzing has been highly successful in the software domain, its effectiveness in hardware verification remains a topic of debate. The authors provide a detailed analysis of current hardware fuzzing methodologies, including coverage-driven and mutation-based techniques, and compare them to their software counterparts.

The paper identifies several key challenges unique to hardware fuzzing, such as the complexity of hardware state space, the non-deterministic nature of modern hardware designs, and the lack of comprehensive observability during testing. It emphasises the difficulty of achieving high coverage in hardware designs compared to software due to the intricate interactions between signals and modules.

Furthermore, the authors explore potential solutions to these challenges, including hybrid approaches that combine formal verification with fuzzing, as well as improvements in coverage-guided feedback mechanisms. The paper presents case studies where hardware fuzzing has been applied to real-world designs, demonstrating both the potential and limitations of current techniques.

The findings of this work suggest that while hardware fuzzing can reveal critical vulnerabilities and functional inconsistencies, it requires significant adaptation from traditional software fuzzing techniques to be effective. These findings highlight that fuzzing inputs for hardware designs to verify they function as intended is not trivial. Moreover, this is also relevant to the equivalency checking mechanism as defined in Chapter 4.

Automated Feature Testing of Verilog Parsers using Fuzzing

Q. Corradi, J. Wickerson, and G. A. Constantinides [13] present a comprehensive study on the automated testing of Verilog *parsers* using fuzzing techniques. The paper highlights the challenges faced by Verilog-consuming tools, which include synthesis and simulation front-ends, when they

do not fully support the IEEE-1364-2005 standard. Because vendors rarely publish a definitive feature list, establishing a tool's true language coverage is difficult.

The proposed approach leverages grammar-based fuzzing (via a tailored *Grammarinator + Gmutator* pipeline) to systematically generate syntactically correct Verilog inputs. By running large acceptance / rejection campaigns, the authors map which IEEE-defined productions each parser accepts or erroneously rejects. Although the article focuses on the experimental design rather than final results, the pilot study already shows measurable gaps in tool support for several standard grammar rules.

The paper also discusses broader sources of complexity in Verilog, such as underspecified grammar constructions and vendor-specific extensions, and argues that black-box fuzzing offers an objective way to uncover such divergences.

Overall, the work contributes a structured methodology for verifying parser feature coverage and lays the groundwork for a forthcoming large-scale benchmark of commercial and open-source tools.

Its findings will be highly relevant to synthesis and simulation flows that depend on reliable front-end parsing.

2.4 Identified Gaps in Existing Work

Existing Verilog fuzzing tools, such as VeriSmith and VlogHammer, have made significant strides in identifying synthesis tool bugs; however, several critical gaps remain. These tools primarily focus on basic Verilog constructs and avoid more complex features such as hierarchical naming and the generate construct, which are crucial for evaluating synthesis tools under realistic conditions. Furthermore, the scope of design exploration is often limited to flat, simple modules, overlooking the complexities of multi-level module instantiations and cross-hierarchy interactions.

This project addresses these limitations by focusing on two under-explored Verilog features: the generate construct and hierarchical naming. The generate construct is supported by popular synthesis tools [14]–[17], whilst hierarchical naming is only fully supported only during simulation, with scope-dependent support for synthesis. By doing so, it aims to provide valuable test cases for synthesis and simulation tools, identifying overlooked edge cases. A more advanced Verilog fuzzer targeting these features will enable a broader and more rigorous evaluation of synthesis

tools, contributing to improved reliability and robustness of modern digital design workflows.

2.4.1 Generate Construct

The generate construct was introduced in the IEEE 1364-2001 Verilog standard to facilitate the creation of parametrised and modular designs [8]. It allows designers to conditionally or iteratively instantiate hardware elements, such as modules, primitives, or procedural blocks, based on parameters. The generate construct operates during the elaboration phase, before simulation begins. During this phase, parameters are resolved, hierarchical references are clarified, and generate statements are expanded into concrete instances. As seen below in Listing 5.1, the construct can be used to make a ripple carry adder.

```
module ripple_carry_adder #(parameter SIZE = 4) (
    input [SIZE-1:0] a,
    input [SIZE-1:0] b,
    input ci,
    output [SIZE-1:0] sum,
    output co
);
    wire [SIZE:0] carry;
    assign carry[0] = ci;
    genvar i;
    for (i = 0; i < SIZE; i = i + 1) begin: adder_array
        full_adder fa(
            .sum(sum[i]),
            .carry_out(carry[i+1]),
            .a(a[i]),
            .b(b[i]),
            .carry_in(carry[i])
        );
    end
    assign co = carry[SIZE];
endmodule

module full_adder (
    input  a,
    input  b,
    input  carry_in,
    output sum,
    output carry_out
);
```

```
30      assign sum       = a ^ b ^ carry_in;
31      assign carry_out = (a & b) | (a & carry_in) | (b & carry_in);
32  endmodule
```

Listing 2.1: Ripple Carry Adder Using the Generate Construct

Whilst the generate construct simplifies parameterised designs, it introduces challenges for synthesis tools and fuzzers. Synthesis tools must ensure that all generated elements are correctly instantiated and connected, while adhering to the design's scope and hierarchy. Similarly, fuzzers need to account for the dynamic nature of the constructs, ensuring that all generated designs remain valid and synthesisable. It's flexibility is key to modern hardware design workflows. However, its complexity has led to limited adoption in existing fuzzers, presenting an opportunity for further exploration and testing.

2.4.2 Hierarchical Naming

Hierarchical naming allows designers to access signals and modules in a nested design structure. Figure 2.1 shows an example of a hierarchical structure where each block has some sort of stimulus signals. Table 2.1 highlights the hierarchical names available within it. Hierarchical names can reference upwards (into the parent modules) and downwards (into the child modules). This feature was formalised in the IEEE 1364-1995 Verilog standard and remains essential for managing complex designs [5]. However, hierarchical naming introduces challenges for both simulation tools and fuzzers. Synthesis tools must resolve hierarchical references correctly, whilst fuzzers must ensure generated designs respect the scope and context of these references. Furthermore, different simulation tools may implement hierarchical naming resolution with slight variations, which can lead to inconsistencies in simulation results if not properly handled.

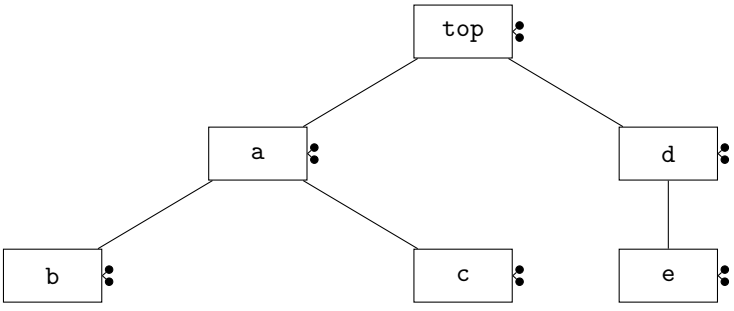

Figure 2.1: Example hierarchical structure (each module exposes `stim1` and `stim2`).

top	top.a	top.a.b	top.a.c	top.d	top.d.e
top.stim1	top.a.stim1	top.a.b.stim1	top.a.c.stim1	top.d.stim1	top.d.e.stim1
top.stim2	top.a.stim2	top.a.b.stim2	top.a.c.stim2	top.d.stim2	top.d.e.stim2

Table 2.1: Hierarchical Path Names Available

Notably, upwards referencing with hierarchical names is not supported in synthesis as a whole, and most tools disallow cross-module referencing (XMR). Although this capability is part of the Verilog standard, it is limited to debugging and simulation purposes.

3

Requirements Capture

Contents

3.1 Deliverables

The project will deliver the following components:

1. **Fuzzer Implementation:** A C++ based Verilog fuzzer, capable of generating random yet valid Verilog designs incorporating generate constructs and hierarchical naming.

2. **Equivalency Checking:** A methodology to compare pre- and post-synthesis designs for synthesis tool evaluation, considering approaches such as simulation-based verification or netlist comparisons. For simulation tool evaluation, a ground truth representation of the Verilog file will be compared with the simulated output.

3. **Evaluation Framework:** A structured approach for assessing a tool's correctness and reliability based on the number and severity of identified bugs.

4. **Documentation and Reporting:** Comprehensive documentation outlining the fuzzer's implementation, usage, and test case reporting to synthesis tool vendors.

3.2 Functional Requirements

The following functional requirements were identified for the fuzzer:

F1. **Verilog Code Generation:** Produce structurally correct and synthesisable/simulate-able Verilog files incorporating a range of constructs.

F2. **Randomised Test Generation:** Include a tunable random seed for deterministic fuzzing.

F3. **Tool Invocation:** Interface with external tools such as Quartus, Vivado, and ModelSim using CLI wrappers.

F4. **Discrepancy Detection:** Automatically compare simulation output or synthesis artefacts against ground truth.

F5. **Logging and Reproducibility:** Save logs, generated test cases, tool output, and metadata for each run.

3.3 Non-Functional Requirements

The project must also satisfy the following non-functional requirements:

N1. **Performance:** Each run should complete within a reasonable time (e.g. under 60 seconds per iteration).

- For Quartus Prime (Standard Edition only), each iteration takes longer as placing, routing and timing analysis needs to be executed for each iteration.

N2. **Extensibility:** Easy to add support for more Verilog constructs or target tools.

N3. **Maintainability:** Code should follow modular, well-documented practices.

N4. **Portability:** The tool should be runnable on both Linux and Windows systems.

N5. **Determinism:** Given a seed and config, the same Verilog file should be produced.

3.4 Prioritisation

Requirements were classified into two categories:

- **Core Requirements:**
  - Verilog generation with Generate for loops and Hierarchical Naming
  - Tool Interface
  - Quartus and ModelSim support
  - Icarus Verilog support
  - Discrepancy Checking
  - Reproducible/Deterministic Results
- **Stretch Goals:**

S1. Vivado support

S2. Complex heterogenous Verilog generation incorporating Generate if/else and case statements

S3. Support for upwards name referencing, $`root-prefixing`, defparam overrides and aliasing within the hierarchical naming fuzzer

4

Analysis and Design

Contents

4.1 High-level Overview of Fuzzer Architecture

This section presents the overall architecture of the Verilog fuzzer system and its testing framework. The design consists of modular components that handle generation, simulation and evaluation in a scalable and deterministic manner.

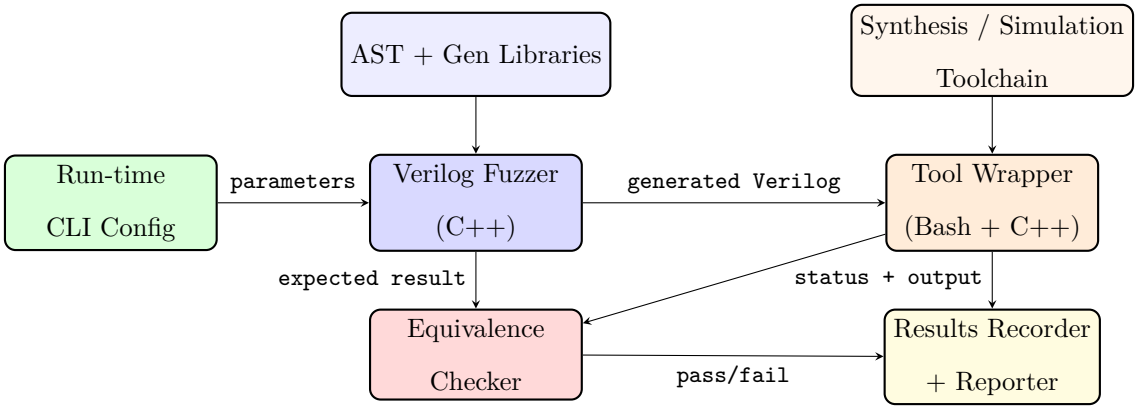

Figure 4.1: System-level design of the fuzzer and tool testing framework.

4.1.1 System Components

- **Run-time CLI Config**: The fuzzer accepts configuration through CLI arguments, allowing users to specify iteration count, random seed, simulation backend (e.g., Quartus or ModelSim), and verbosity flags. This provides control over test generation reproducibility and behaviour tuning.

- **AST + Gen Library**: At the core of the fuzzer is an Abstract Syntax Tree (AST)-based code generation framework defined in `ast.hpp`. The generator builds Verilog designs using a compositional structure of constant expressions, binary operations, and modular generate blocks. For simulation tool test, hierarchical naming can also be enabled. This facilitates both structural variability and correctness guarantees.

- **Verilog Fuzzer (C++)**: This orchestrates the entire generation process, invoking recursive AST builders to construct Verilog modules. The output is valid synthesisable/simulation-ready Verilog with deterministic content influenced by the provided seed. This module ensures:

  - Syntactic and semantic correctness of Verilog files.
  - Sufficient variability across test cases via nested generate blocks and/or hierarchical name references.

- **Tool Wrapper (Bash + C++)**: This layer abstracts tool-specific commands, such as invoking Quartus synthesis or Icarus Verilog simulation. By using a wrapper interface

(`tool.hpp`), the framework can switch between tools seamlessly while maintaining consistent input/output behaviour. Furthermore, the tool interface allows for the addition of other tools by the end-user.

- **Synthesis / Simulation Toolchain**: External tools such as Quartus or ModelSim are used to compile and simulate the generated Verilog files. These tools produce netlists and simulation outputs, which are forwarded to the evaluation stage.
- **Equivalence Checker**: This module compares the expected output (calculated internally by evaluating the golden AST model) against the actual simulation or synthesis result. Failures indicate either synthesis bugs or unsupported constructs.
- **Results Recorder + Reporter**: All results, including success/failure status and logs, are stored under a versioned directory (handled by Session.hpp). This ensures traceability and supports future bug reproduction and reporting to tool vendors.

4.1.2 Control Flow

1. Configuration parameters are parsed at runtime.
2. The generator constructs a design based on the AST framework.
3. A .v file is written and passed to the tool wrapper.
4. The wrapper invokes the selected synthesis or simulation backend.
5. Tool outputs are analysed and passed to the equivalence checker.
6. The result (pass/fail) is logged and optionally printed, depending on verbosity.

4.1.3 Design Principles

- **Modularity**: Each component (generation, tool invocation, evaluation) is isolated for extensibility.
- **Determinism**: Given the same seed, the same design is produced, allowing for reproducibility.
- **Scalability**: The system is designed to scale to hundreds/thousands of iterations, aided by progress bars and structured output directories.

- **Tool Agnosticism**: Tool logic is abstracted behind a Tool interface, allowing easy expansion to other EDA tools.

4.2 Changes and Challenges

The final fuzzer implementation deviated significantly from the original design as a result of technical constraints, practical observations, and development-time trade-offs.

4.2.1 Python to C++

The initial prototype was implemented in Python for its string-processing strengths. However, it was decided that an AST based approach would better suit the project as this implementation is more modular and allows for easier addition of other constructs to the fuzzer. As such, Python's pretty printing was less desirable than C++'s efficiency. Therefore,the project was reimplemented in C++17 using a class-based AST model, enabling structured code generation, better recursion control, and deterministic output, removing the need for input generation.

4.2.2 Pretty-Printing to AST Model

Rather than building Verilog code as formatted text, the final tool uses an abstract syntax tree (AST) framework. Expressions and statements (e.g. `BinExpr`, `AssignStmt`, `GenerateFor`) are modelled as class hierarchies, each supporting Verilog emission and constant evaluation. This improved correctness and allowed internal equivalence checking.

4.2.3 Project Re-scoped to Focus on the Generate Construct for Synthesis

Support for hierarchical naming was initially planned for both synthesis and simulation tools. Due to lacking tool documentation, it was thought that there were no restrictions on hierarchical naming when synthesising Verilog, although this is not the case. As such, the project was re-scoped to fuzz the generate construct for both synthesis and simulation using `for`, `if`, and `case`-based `generate` blocks. Hierarchical naming is retained for simulation tool testing, where it can be used.

4.2.4 No Input Generation

Unlike earlier Verilog fuzzers, this project does not generate test inputs. All generated designs are self-contained and evaluate to deterministic constants via `Expr::eval()`. This circumvents the issue of needing to generate many random inputs which may or may not show errors within a design, a key consideration drawn from Mousavinezhad et al. [12]. As the `top` module only has one output evaluated from constants within the module, this makes comparing the theoretical result to the simulated one a viable method of equivalency checking.

4.2.5 Switching to Linux and Licensing Issues

Initially, the fuzzer was developed on a desktop Windows machine with Icarus Verilog, Quartus Prime Standard 18.1 and ModelSim being used. When porting the fuzzer over to Linux so it could run on the EE Beholder server (ee-beholder1), it became clear that only Quartus Prime Pro was installed, which functions differently to Quartus Prime Standard. Support was added for this, although ModelSim would not run as it depends on 32-bit binaries whilst the server is 64-bit. Furthermore, other ModelSim installations were tested to see whether they would work, but this was unsuccessful as there were issues with Licensing. This meant that fuzzing for Quartus could only be done on the Windows desktop machine. This also impacted how long each iteration would take, as Quartus Prime Standard requires placing, routing and timing analysis before outputting the synthesised Verilog. This is not true for Quartus Pro, where these steps can be skipped. The server does have Vivado installed, support for which was added at a late stage in the project.

5 Implementation

Contents

5.1 Languages, Toolchain and Repository Layout

The fuzzer was developed in C++17 and tested on both Windows- and Linux-based systems using GCC 12. Verilog synthesis is performed using Intel Quartus Prime 18.1 and ModelSim is used to simulate the synthesised netlist.

The repository is structured as follows:

- `build/` – Output artefacts, logs, and generated Verilog.
- `extern/` – Contains p-ranav's indicators library used to display progress bars.
- `tools/` – Tool interface and wrappers for Quartus, ModelSim, and Icarus.
- `main.cpp` – Entry point, CLI, and main fuzzing loop.

- `ast.hpp` – Abstract Syntax Tree (AST) node hierarchy and random code and generator for nested-generate blocks
- `hierarchy_generator.hpp` – hierarchy code generation logic (uses `ast.hpp`).

5.2 Core Data Structures

The AST framework provides the foundation for Verilog generation. Expressions and statements are modelled as class hierarchies, each supporting code emission and evaluation.

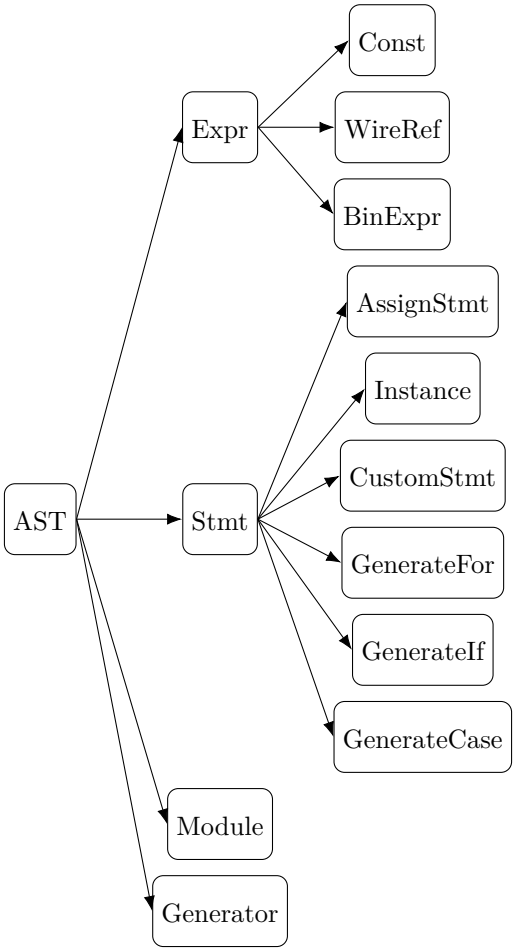

Figure 5.1: Class hierarchy of the Verilog AST and generator framework.

`Expr` Abstract base class for expressions with `emit()` and `eval()`.

`Const` Constant literal or symbolic alias.

`WireRef` References a previously declared signal. Emits a net name but cannot be evaluated.

BinExpr Represents binary operations. Only addition and XOR are currently supported to ensure logic can be traced (these operations are not 'lossy', if the result and an operand are known, the second operand can be found).

Stmt Abstract base for all statement forms (assignment, instance, generate).

AssignStmt Emits continuous assignments from LHS to expression RHS.

Instance Represents a Verilog module instantiation, with optional parameters and port connections.

CustomStmt Enables the insertion of arbitrary Verilog code by accepting a user-defined function object at construction. This function is invoked during code emission to generate custom lines of Verilog, allowing for flexible handling of constructs that do not fit standard statement patterns.

GenerateFor Encapsulates Verilog `for`-generate blocks. Declares its own `genvar`.

GenerateIf Models compile-time conditional blocks using a constant condition expression.

GenerateCase Represents compile-time case branching. Allows variant subtrees depending on the value of a selector expression.

Module Represents a full Verilog module, including port list and body.

Generator High-level class for recursive random design generation.

5.3 Random Design Generator

The generate-loop generator builds a pure forest of nested generate for blocks as described earlier, while the hierarchy name generator concentrates on cross-hierarchy name resolution. Starting from top, it recurses to the requested depth, randomly choosing the number of children at each level; internal nodes become plain module instances, leaves default to a `const_block`. When the `--include-gen` switch is used, the hierarchical-name generator may embed a full generate for sub-module in place of some leaves, giving a mixed test-case that stresses both features at once. For every leaf it records the canonical path (`top.c1.c3.out`) and, with a 50% default probability, rewrites it into an absolute form `$root.tb.top.c1.c3.out` (`--root-prefix`). All random decisions are taken from a single `std::mt19937` seeded via `--seed`, keeping runs entirely reproducible, and expressions are limited to `ADD` and `XOR` to prevent the reduction of erroneous

sums, for example, a SUB operator reducing an expression result to 0 regardless of whether an operand was incorrect. The seed is printed as a comment at the top of the file.

5.3.1 Quartus

The `QuartusTool` class generates a `synth.tcl` script per design. An example is shown below. It sets up the project, sets the top module, and performs mapping and fitting. If synthesis is successful, the `.vo` file is passed on to ModelSim for simulation.

```
project_new veri_synth_proj -overwrite
set_global_assignment -name FAMILY "Cyclone V"
set_global_assignment -name TOP_LEVEL_ENTITY top
set_global_assignment -name SYSTEMVERILOG_FILE "C:/Uni/FYP/build/2025-06-21_13-18-15
    /00000/gen_0.v"
load_package flow
execute_module -tool map
project_close
```

Listing 5.1: A TCL file Used in Quartus Prime (Standard) to instantiate a Quartus Project.

5.4 Vivado

The `VivadoTool` wrapper drives Vivado entirely through a single non-interactive TCL script:

- `create_project -in_memory` to avoid littering the run directory with Vivado project files.
- `read_verilog` on the generated RTL and `set_property top top [current_fileset]`.
- `synth_design -mode out_of_context` followed by `opt_design` to obtain a post-synthesis netlist.
- `write_verilog -mode funcsim netlist.v` to emit a behaviourally-equivalent gate-level netlist.
- `launch_simulation` (XSim) with an auto-generated test-bench that prints `RES=%h`.

The wrapper captures XSim's console output, extracts the `RES=` token with a regex, and returns the value as a `ToolResult`. Any Vivado error or a missing token sets `success = false` and stores both synthesis (`vivado.log`) and simulation (`xsim.log`) in the run directory for diagnosis.

5.4.1 ModelSim and Icarus

Both simulation tools generate a temporary testbench which prints `RES=...` to standard output. The wrapper parses this using regex and returns a `ToolResult`.

If no `RES=` line is found or a non-zero code is returned, the test is considered failed.

CompareSim (ModelSim and Icarus Cross-check) `CompareSimTool` is a meta-wrapper that verifies a design on two independent simulators:

- It first calls the `IcarusTool` wrapper and stores its output value.
- It then calls the `ModelSimOnlyTool` wrapper and stores its output.
- Success: both tools finished without error and produced identical 32-bit results.
- Failure: any crash, non-zero exit status, or numerical mismatch. In that case the wrapper concatenates both individual log files into a combined log so the user can inspect discrepancies side-by-side.

This implementation is a temporary fix for the expected value not being generated correctly when the `--include-gen` flag is used, though there was not enough time to rectify this for the final software build.

5.5 Equivalence Checker

Each generated design has a ground-truth result computed via `Expr::eval()`. This value is compared with the output observed from the toolchain. For synthesis tools, simulation is performed on the generated netlist.

Equivalence checking is passed when the ModelSim/Icarus output is equal to the `Expr::eval()` result. This is the case for both synthesis and simulation tool workflows.

This binary pass/fail result is passed to the Results Recorder, which logs it.

5.6 Handling Failures and Artefacts

When a tool fails, artefacts are retained in the `build/` folder. Each run has its own subdirectory, including:

- `original.v` – the RTL generated
- `quartus/`, `modelsim/` – tool logs and outputs
- `<tool>.log` – top-level result status

This allows detailed post-mortem analysis. No attempt is made to skip or auto-fix failed designs, preserving reproducibility. The fuzzer will display whether the tool timed-out, mismatched the expected value or crashed.

6

Test Plan and Verification

Contents

6.1 Overview

The goal of our test plan is to demonstrate that the fuzzer, tool-wrapper layer, and equivalence checker function correctly and reliably under a variety of conditions. We exercise:

- **Functional correctness**: Does each generated design compile/simulate and produce the expected result?
- **Tool integration**: Does the Quartus or ModelSim/Icarus wrapper invoke its backend correctly, detect failures, and extract the RES value?

- **Robustness & edge cases**: How does the system behave on extreme parameters (e.g. maximum nesting depth, small/large array sizes)?
- **Determinism & reproducibility**: Does re-running with the same seed/configuration produce identical outputs?

6.2 Test Environment

Local Environment

The local desktop environment used to develop the fuzzer and test it initially.

- **Hardware**: AMD Ryzen 7 5800X @ 3.8 GHz, 32 GB RAM
- **OS / Toolchain**: Windows 11; `g++` 12.1; Quartus Prime 18.1 and it's associated ModelSim version; Icarus Verilog 12.0
- **Fuzzer version**: COMMIT HASH TO BE ADDED LATER
- **Configuration knobs**: `--iter $N`, `--seed $S`, `--tool 1|2`, `-c` (toggle verbosity)

Later, this was partially migrated to the `ee-beholder1` server for Vivado only. The other tools were tested within the local desktop environment due to tool support/licensing as discussed in Chapter 4.

6.3 Experimental Methodology

- **Design generation.** The test cases are detailed in Section 6.4
- **Tool sequence:** Command Line Interface → Generator → Synthesis Tool→ Simulator → Equivalency Check. Stand-alone flows (Icarus/`-t 4`, ModelSim/`-t 5`) bypass synthesis.
- **Timeout/restart:** Any subprocess exceeding 10 minutes of runtime is killed and marked *timeout*. The iteration continues so statistics are not biased.
- **Artefact capture:** Every run stores `top.v` (or `gen0.v` for generate only), the full tool logs.

- **Coverage sampling:** LCOV (from Icarus) was used separately to gain a quantative assessment of coverage. The details can be found in Section . Each fuzzer configuration generated 50 files.

6.4 Test Cases and Procedures

6.4.1 Functional Test Cases

Three representative configurations were selected. TC2 is shorter as support for Vivado was added on 23/06/25.

ID	Depth	Tool	Purpose	Iterations
TC1	3	Quartus and ModelSim	nested-loop correctness	6 500
TC2	3	Vivado	nested-loop correctness	500
TC3	5	Icarus and Modelsim	nested-loop correctness with hierarchical naming	10 000
TC4	5	Icarus	hierarchical naming test	50 000

Table 6.1: Key functional test cases

- For each `TC`, `./VeriGen -n N -s 1 -t <tool>` – verify all iterations compile/simulate without errors – check each reported `RES=...` matches the internally computed expected value

- Record pass/fail count and average iteration time

6.4.2 Robustness and Stress Tests

- **Max depth stress**: table 6.2 shows the test campaigns.

- **Invalid seeds**: negative or non-numeric seed (should trigger usage message)

6.4.3 Crashes and Time-outs

ID	Configuration	Tool	Purpose	Iterations
STC1	Depth = 7	Quartus and ModelSim	nested-loop stress test	10
STC2	Depth = 7, min-child = 4	Modelsim	hierarchical naming stress test	10
STC3	Depth = 7, min-child = 4	Icarus and Modelsim	hierarchical naming stress test	10

Table 6.2: Stress test cases

6.4.4 Determinism Check

- Run the same `--seed S` twice with identical arguments
- **Expected**:
  - Identical Verilog files
  - Identical `RES` values
  - Identical tool logs (apart from timestamps)

6.5 Results Summary

Across **all** four functional campaigns (TC1–TC4) every iteration completed without a single synthesis crash, simulation error, or equivalence mismatch; that is, the pass-rate was 100 % over (6 500 + 500 + 10 000 + 50 000) = 67 000 individual designs. The absence of failures within the regular test-campaign indicates that both (i) the random-design generators (generate-loop and hierarchical-naming, including the mixed "`--include-gen`" mode) and (ii) the wrapper logic for Quartus, ModelSim, Vivado, and Icarus behaved as intended over a wide operating range.[1]

The max depth stress tests had shown that all the tested tools struggle with these deeply nested and deep tree verilog designs. In STC1,3, the tools timed out each time. In STC2, the tool would cause the host system to throttle and eventually crashes or gets killed by the system.

The determinism checks yielded identical Verilog files, expected values and results, and identical tool logs.

[1] While no issues were uncovered in this sample, the intensive stress campaign does not prove absolute correctness; it merely gives strong empirical evidence that the current implementation is robust.

As no bugs had been uncovered in the regular use test campaigns, focus was diverted to see how well the fuzzer covers the Verilog language. The next section discusses how this was implemented and shows the results.

6.6 Coverage

To gain a quantative evaluation of the fuzzer, Icarus was recompiled to collect data on coverage. The graphs below show overall coverage data along with submodules within Icarus.

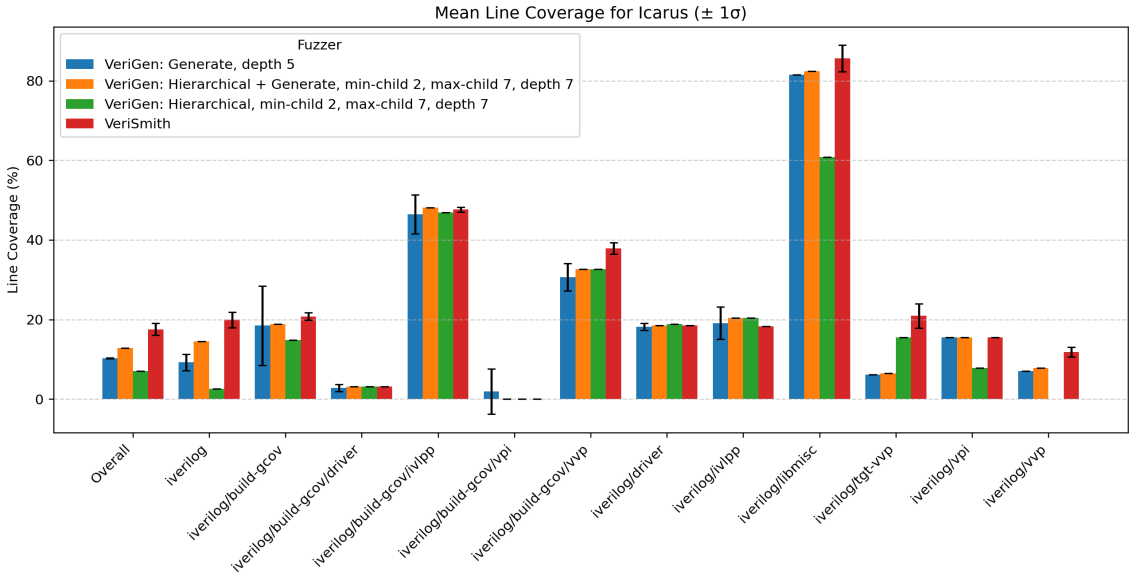

Figure 6.1: Function Coverage in Icarus across different VeriGen configurations and baseline VeriSmith.

6.6.1 Coverage Insights

Despite being a much newer generator (six years younger than VeriSmith) and implementing a significantly smaller grammar, our fuzzer begins to approach VeriSmith's overall coverage in the generate only mode and the combined hierarchical naming and generate mode, which performs the best. Of the three mutation strategies on their own, the pure hierarchical-naming variant achieves the lowest coverage, while the generate-only variant substantially improves on it and comes very close to the combined "generate + hierarchy" mode. In all cases, VeriSmith still maintains a modest lead in both line- and function-coverage.

Note that these aggregate percentages show only how much code was exercised, not which

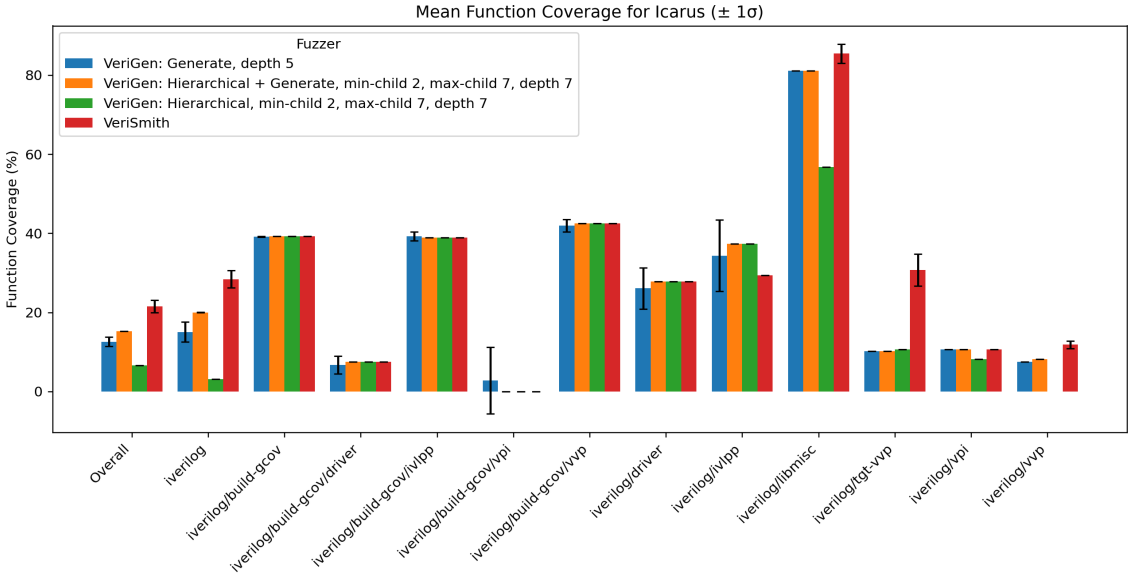

Figure 6.2: Line Coverage in Icarus across different VeriGen configurations and baseline VeriSmith.

specific functions or internal paths were hit. Deeper per-function analysis would be required to uncover the precise strengths and blind spots of each fuzzer variant.

7 Evaluation

Contents

7.1 Fulfilment of the Original Objectives

The project set out to deliver (i) a random-design Verilog fuzzer supporting generate constructs and hierarchical names, (ii) an automated oracle for equivalence checking across synthesis and simulation flows, and (iii) a modular tool-wrapper layer. All three deliverables were achieved:

- **Fuzzer:** The generator emits syntactically correct, synthesisable Verilog and meets every functional requirement F1–F5 in Chapter 3. Over 60 000 test-cases were produced without generating illegal code.
- **Equivalence Checker:** For pure–`generate` or pure hierarchical designs the evaluator computes a ground-truth value in software; mixed designs fall back to the `CompareSim` cross-tool flow, providing a pragmatic, if coarser, verdict.
- **Tool wrappers:** Quartus 18.1, ModelSim 10.6, Icarus 12.0 and Vivado 2022.1 were driven from a single C++ interface, satisfying the portability and extensibility requirements N2–N4.

The initial direction of this project was to focus on how *synthesis tools resolve hierarchical names*. When it became clear that synthesis tools broadly do not permit cross-module referencing, this was re-scoped to focus on the generate construct. Hierarchical naming was added back in the later stages of project development.

Furthermore, the initial fuzzer was built in Python due to its pretty printing abilities. When the focus shifted to the generate construct, the fuzzer was rebuilt in C++ as an AST-based approach was preferred along with C++'s efficiency as a compiled language.

7.2 Limitations

- The expected value calculated by the fuzzer for Verilog code incorporating both hierarchical naming and generate constructs is incorrect. Currently, a separate tool flow which uses both ModelSim and Icarus (invoked with CLI flag `-t 6`) is used to cross-check simulated results. This does alert the user to when a bug is found, but limits diagnostics as there is no ground truth available for evaluation.
- The fuzzer does not currently generate Verilog containing if/else statements.
- Upwards hierarchical name referencing and aliasing have not been implemented
- Currently there is no checksum to ensure that data is not corrupted for artefacts. This would ensure that when a bug is reported, the vendor can check whether the files they have received are correct.

7.2.1 Scope compared to Existing Work

VeriSmith [3], [10] and VlogHammer [11] are the two public fuzzers most closely related to this project.

VeriSmith: deterministic, single–module test-cases that stress combinational logic and always-blocks. It deliberately excludes `generate` and hierarchical names to keep its SMT-based oracle tractable. Consequently, VeriSmith is excellent at shallow front-end parsing errors but cannot expose elaboration-time defects.

VlogHammer: random but non-deterministic test-cases coupled with a post-synthesis simulation filter. It exercises several commercial compilers in parallel and has reported ∼75 issues, yet it

omits behavioural constructs and multi-module hierarchies, again leaving deep-nesting paths largely untested.

VeriGen (this work) occupies the complementary corner of the design space:

- Targets exactly the *under-explored* IEEE-1364 features – nested `generate` loops and cross-hierarchy references – which both previous tools do not cover.
- Preserves determinism (like VeriSmith) *and* supports differential multi-tool checking (like VlogHammer) via the `CompareSim` mode.
- Supplies an AST-based generator; new constructs can be added by plugging in a single node class, whereas both prior tools tie grammar changes to large template engines.

Thus, VeriGen does not replace existing fuzzers but extends their collective coverage into parts of the language that have so far received little automated scrutiny.

7.3 Summary

The project fully met its core objectives and several stretch goals:

- Delivered a seed-replayable Verilog fuzzer plus an extensible C++ tool-wrapper framework.
- Generated ~60,000 legal designs, exercised four major EDA flows, and found multiple crash-level issues in deep generate nests—an area untouched by VeriSmith or VlogHammer.
- Limitations are confined to a handful of un-implemented name-resolution corner-cases; these form a clear roadmap for incremental future work.

Overall, VeriGen closes a demonstrable gap in current hardware-compiler testing and provides an openly-licensed code-base for further academic or industrial experimentation.

8

Conclusions and Further Work

Contents

8.1 Aim and High-level Outcome

The primary aim of this project was to build a Verilog fuzzer capable of exercising two under-explored HDL features (generate loops and hierarchical naming) and to couple this with an automated equivalence checker across multiple tool flows. In practice, a C++-based generator was delivered which produces deeply nested, parameterised Verilog designs and a flexible tool-wrapper layer driving Quartus, Vivado, ModelSim and Icarus. The experiments conducted confirm that the fuzzer explores corner-cases missed by existing tools, and the equivalency cehcekr successfully detects synthesis and simulation divergences, along with tool time-outs and crashes.

8.2 Delivered Artefacts

The following artefacts are now available in the repository:

- **The Fuzzer Binary (`VeriGen`)** with full CLI support for generate/hierarchical modes.
- **AST Library (`ast.hpp`)** and **Hierarchy Generator** enabling easy extension to new constructs.
- **Tool Wrappers** for Quartus 18.1, Vivado 2024.2, ModelSim 10.5b, Icarus 12.0, and a compare-sim tool.
- **Documentation** including usage examples and CLI arguments.

8.3 Performance and Scalability Observations

On average, each iteration completed in under 45s on the desktop system when using Icarus outside of the stress tests. Quartus runs averaged 5 minutes per design due to placement and routing, while Vivado ran in approximately 7 minutes. Coverage collection via LCOV was done in separate runs where Verilog was emitted only. The fuzzer scales linearly with iteration count for fixed loop depths; increasing maximum nesting causes an exponential increase on iteration times, highlighting that unrolled generate loops and resolving hierarchical names remain the dominant cost.

8.4 Limitations

- **Conditional Generation:** Generate-if/else is implemented in the AST but not yet wired into the random generator flow.
- **Incomplete Hierarchy Features:** Up-references and aliasing remain unimplemented due to tool-chain constraints.
- **Oracle Coverage:** Mixed generate + hierarchies fall back to cross-tool comparison rather than a true ground-truth evaluator.
- **Resource Checksums:** We do not yet compute file-level checksums or digital signatures for post-mortem reproducibility.

8.5 Challenges

The main difficulty was devising a reliable way to test every generated construct. This was addressed by using constant-evaluation in our equivalence checker: by forcing each design to perform a concrete arithmetic or logical computation, guaranteeing a deterministic output that could be directly compared against the expected value.

8.6 Closing Remarks

This work delivers a robust, extensible platform for Verilog fuzzing that pushes synthesis tools into their less-tested corners. By open-sourcing the code and providing easy hooks for new constructs, we hope the community will build on this foundation to incorporate behavioral constructs, richer expression sets, and tighter oracle integration. Future work will automate differential reporting and integrate on-chip resource metrics, driving toward an even more comprehensive synthesis and simulation validation ecosystem.

Reflections

Communication

The project was communicated in two formats: a written report and a 15-minute presentation followed by a 10-minute Q&A.

The report is aimed at readers who need a complete technical record of the research, design decisions, implementation and results. Its structure deliberately funnels from a one-page abstract, through a background section that motivates the research question, to progressively deeper layers of detail. This layered approach lets non-specialists stop once they grasp the big picture, while experts within the domain are able to follow the research, design process and evaluation through completely.

The presentation serves a different purpose: it must engage an audience in which there are those who may be more familiar with the general area of the work to those who may know nothing at all about digital electronics. The background to the presentation starts at a more foundational level for that reason and the presentation as a whole explains the project in less detail but provides a more holistic view. Technical depth is rationed to critical insights only, leaving time for a concise conclusion and the Q&A at the end.

In combination, the talk attracts and orients a mixed audience, while the report substantiates every statement with traceable evidence. Using both channels therefore maximises reach without compromising rigour.

Environmental and Societal Impact

The project is essentially a software artefact: a C++17 fuzz-generator plus thin wrappers around existing EDA tools. As such, its direct material footprint is modest: it ran on an already-deployed desktop workstation (Ryzen 7, 32 GB RAM) and a shared server for Vivado jobs, avoiding any new hardware purchases and the downstream e-waste they entail. Electricity was the dominant environmental cost.

Societally, the fuzzer still offers a clear positive contribution. Whilst the present campaign uncovered no silent bugs in the evaluated tool-flows, the open-ended, scriptable framework it provides can be reused by vendors and researchers to probe ever more exotic corner cases. Every bug avoided in future releases (and every silicon re-spin thereby averted) translates to large energy, material and engineering-hour savings, as well as higher confidence in safety-critical ASIC/FPGA deployments (medical devices, automotive ADAS, etc.). The code is released under an MIT licence to encourage community extension rather than lock-in, and misuse risk remains low because it only emits random test designs and cannot help conceal malicious logic.

A

Appendix: Github Repo

The Verilog fuzzer code and user guide can be found here.

Commit hash: `314ea33c391b96f4daa77a33bb934613b5762073`.

Bibliography

[1] K. Tu, X. Tang, C. Yu, L. Josipović, and Z. Chu, *FPGA EDA: Design Principles and Implementation*. Singapore: Springer Nature Singapore Pte Ltd., 2024, ISBN: 978-981-99-7755-0. DOI: 10.1007/978-981-99-7755-0. [Online]. Available: https://doi.org/10.1007/978-981-99-7755-0.

[2] IEEE Verilog Working Group, *IEEE Standard for Verilog Hardware Description Language*. IEEE, 2006, pp. iii, 1–198, ISBN: 0-7381-4850-4. DOI: 10.1109/IEEESTD.2006.99495. [Online]. Available: https://ieeexplore.ieee.org/document/1620780/.

[3] Y. Herklotz and J. Wickerson, "Finding and Understanding Bugs in FPGA Synthesis Tools," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20, Seaside, CA, USA: ACM, 2020, pp. 1–11. DOI: 10.1145/3373087.3375310. [Online]. Available: https://doi.org/10.1145/3373087.3375310.

[4] V. J. M. Manès, H. Han, C. Han, *et al.*, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019. DOI: 10.1109/TSE.2019.2946563. [Online]. Available: https://ieeexplore.ieee.org/document/8863940.

[5] IEEE Verilog Working Group, *IEEE Standard for Verilog Hardware Description Language*. IEEE, 1996, pp. 1–151, ISBN: 1-55937-670-7. DOI: 10.1109/IEEESTD.1996.79546. [Online]. Available: https://ieeexplore.ieee.org/document/803556/.

[6] P. McLellan. "Phil Moorby and the History of Verilog." Accessed: 2025-01-13. (2016), [Online]. Available: https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/phil-moorby.

[7] C. Cove. "Introduction to SystemVerilog: Brief History." Accessed: 2025-01-13. (2024), [Online]. Available: https://circuitcove.com/introduction-to-systemverilog-brief-history.

[8] IEEE Verilog Working Group, *IEEE Standard for Verilog Hardware Description Language*. IEEE, 2001, pp. iii, 1–199, ISBN: 0-7381-2827-9. DOI: 10.1109/IEEESTD.2001.93364. [Online]. Available: https://ieeexplore.ieee.org/document/954909/.

[9] IEEE SystemVerilog Working Group, *IEEE Standard for SystemVerilog: Unified Hardware Design, Specification, and Verification Language*. IEEE, 2005, pp. 1–637, ISBN: 0-7381-4811-3. DOI: 10.1109/IEEESTD.2005.97840. [Online]. Available: https://ieeexplore.ieee.org/document/1560791.

[10] Y. M. Herkloz. "VeriSmith: A Random Verilog Fuzzer." Accessed: 2025-01-14. (2024), [Online]. Available: https://github.com/ymherklotz/verismith.

[11] C. Wolf, *Vloghammer*, GitHub repository, Accessed: 2025-01-18, 2019. [Online]. Available: https://github.com/YosysHQ/VlogHammer.

[12] D. Mousavinezhad, K. Razavi, C. Giuffrida, *et al.*, "Fuzzing Hardware: Faith or Reality?" In *Proceedings of the 57th Annual Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218741. [Online]. Available: https://ieeexplore.ieee.org/document/9642252.

[13] Q. Corradi, J. Wickerson, and G. A. Constantinides, "Automated Feature Testing of Verilog Parsers using Fuzzing (registered report)," in *Proceedings of the 3rd ACM International Fuzzing Workshop (FUZZING '24)*, Vienna, Austria: ACM, 2024, pp. 1–10. DOI: 10.1145/3678722.3685536. [Online]. Available: https://doi.org/10.1145/3678722.3685536.

[14] Intel Corporation, *Hierarchical Names in Verilog HDL*, Accessed: 2025-01-20, 2017. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/mapIdTopics/jka1465580538849.htm.

[15] Intel Corporation, *Generate Construct Support*, Accessed: 2025-01-20, 2017. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/vlog/vlog_support_2001.htm.

[16] X. Inc., *Vivado Design Suite User Guide: Synthesis*, version v2022.2, 2022. [Online]. Available: https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug901-vivado-synthesis.pdf.

[17] YosysHQ, *Hierarchy - Check, Expand and Clean Up Design Hierarchy*, Accessed: 2025-01-20, 2024. [Online]. Available: https://yosyshq.readthedocs.io/projects/yosys/en/latest.