Finding and understanding bugs in FPGA place-and-route engines

Ollie Cosgrove
oliver.cosgrove21@imperial.ac.uk
Imperial College London
London, UK

Alastair F. Donaldson
alastair.donaldson@imperial.ac.uk
Imperial College London
London, UK

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

Abstract

Place-and-route (P&R) engines are a crucial part of the FPGA design flow, responsible for mapping a given netlist onto the physical resources of an FPGA device. It is critical that they work correctly, because they run after most verification activities have completed, and because their opaque machinations produce complicated outputs that defy manual inspection. However, we know of no prior work that scrutinises the correctness of P&R engines directly.

We present a novel fuzzing-based technique for identifying bugs in FPGA P&R engines, implemented in an open-source tool called FUZNET. FUZNET generates random netlists and feeds them to P&R engines, with the aim of finding cases where the pre- and post-P&R designs are not functionally equivalent. The design of FUZNET incorporates novel solutions that overcome the challenges of (1) generating designs that are complex enough to trigger bugs but small enough to enable high testing throughput, (2) finding a rapid way to check pre- vs. post-P&R equivalence, and (3) automatically reducing suspicious test-cases ready for concise bug reports. We have used FUZNET to generate 57,316 random netlists for testing AMD's Vivado P&R engine. About 3.3% of them failed pre- vs. post-P&R equivalence, and after test-case reduction, we isolated two functional bugs in logic optimisations carried out during P&R (in CFGLUT5 retargeting and in CARRY4 constant propagation) one of which has been confirmed and fixed by AMD.

Our work demonstrates that although the nonfunctional properties of P&R engines make them challenging targets for traditional fuzzing techniques, it is possible to design strategies to overcome these challenges, and thus obtain a valuable fuzzing technique that can identify subtle bugs in mature P&R engines, and ultimately help to make them more trustworthy in safety-critical settings.

CCS Concepts

• **Hardware → Electronic design automation**; **Reconfigurable logic and FPGAs**; *Equivalence checking*; • **Software and its engineering** → *Software testing and debugging*.

Keywords

equivalence checking; fuzzing; logic optimisation; placement; routing; test-case reduction

ACM Reference Format:
Ollie Cosgrove, Alastair F. Donaldson, and John Wickerson. 2026. Finding and understanding bugs in FPGA place-and-route engines. In *Proceedings of the 2026 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '26), February 22–24, 2026, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3748173.3779185

1 Introduction

Place-and-route (P&R) engines are responsible for mapping a digital circuit, given in the form of a netlist, onto the physical resources available on a field-programmable gate array (FPGA). The 'placement' stage determines where on the chip to put each logical element of the design, while the 'routing' stage decides how to connect them together. Several P&R engines are available to designers, some of which are built into commercial synthesis tools such as Altera Quartus Prime [2] and AMD Vivado [4], and some of which are open source (e.g. nextpnr [23] and VPR [9]).

It is crucial to have robust techniques for ensuring that P&R works correctly, for two main reasons: because P&R is *complicated* (and hence at risk of containing bugs) and because it is *trusted* (and hence any bugs it does contain could cause significant problems).

- **P&R is complicated.** This is partly because both placement and routing have huge search spaces. But in addition to this inherent complexity, P&R engines often optimise the circuit itself during P&R. For instance, Vivado's optimisations include logic replication, constant propagation, and LUT merging [4].
- **P&R is trusted.** P&R runs late in the hardware synthesis flow, after most verification activities have concluded, and thus there are few opportunities to catch any bugs it may have introduced. Moreover, the output of P&R is usually large and in an opaque, low-level format, and hence difficult for humans to inspect manually.

Current recommended practice for *vendors* of P&R engines is to have their code certified as conforming to the relevant international standards. For instance, AMD Vivado has been certified to conform to ISO 26262 and IEC 61508 [26], which specify requirements of software tools used in the design of safety-critical electronics. However, certification is neither necessary nor sufficient for ruling out all bugs.

Meanwhile, current recommended practice for *users* of P&R engines is to check each *instance* of P&R for correctness. For example, the NASA Programmable Logic Devices Handbook recommends *back-annotated simulation* [19, §8.4.1.4], where the pre-P&R design is simulated using post-P&R timing information. However, timing-accurate simulation is time-consuming, and the thoroughness with which the post-P&R design is tested is entirely dependent on the exhaustiveness of the user-provided testbench.

The main contribution of this paper is a new method for directly testing the correctness of P&R engines. Other works have studied the correctness of other EDA tooling, e.g. high-level synthesis [12, 17], logic synthesis [14, 16, 22, 28, 33], simulation [24, 25, 27, 32] and formal equivalence checking [20], but we believe ours is the first work to focus on the correctness of P&R specifically. Our method is based around equivalence-checking the pre- and post-P&R netlists, which means it is able to detect bugs where the P&R engine has optimised the input netlist incorrectly, but it cannot check for bugs in the actual placement and routing decisions (such as accidentally mapping two nets to the same multiplexer).

1.1 Why testing P&R engines is challenging

At first sight, randomized testing of P&R engines might seem like a straightforward application of existing technology – e.g. we could use an RTL generator such as Verismith [14] to generate a stream of interesting designs, push these through logic synthesis to obtain netlists, and check in each case that the output of P&R is equivalent to the input netlist. However, such a naïve approach is a non-starter for several reasons:

1. High latency. P&R is a slow process. This means that our testing throughput will be markedly slower than for, say, conventional compilers. Hence, we need to make every test-case as valuable as possible. It also means that our test-case reduction algorithm (an essential part of any randomised testing setup) needs to be designed so that it converges as rapidly as possible.

2. Coverage of legal netlists. A logic synthesis tool will only produce a subset of the valid netlists that a P&R engine can accept. This subset will have been well tested already. We are more likely to find bugs in a P&R engine by providing it with netlists that are *not* necessarily in this subset, yet are still valid.[1]

3. Limit on test-case size. P&R test-cases can't be too big. For conventional compilers, it is known that "larger test cases expose more compiler errors" [29], but for P&R, a test-case whose netlist is clearly too big to fit on the target device is useless – the P&R engine would simply waste time performing futile optimisations trying to make it fit. As such, our test-case generation algorithm must be designed so that the size of netlists is carefully controlled.

4. Constraint-guided optimisations. Conventional compilers aim to produce the best-possible target code, but EDA tools are guided by fixed timing and resource constraints, and do not apply further optimisations once those are met. This makes the usual test-case reduction process more challenging, because if removing parts of a failing test-case makes the bug goes away, we don't know whether this is because we removed the bug-triggering part of the test-case or simply because the smaller test-case required fewer optimisations to meet the constraints.

5. Lack of reference. Our testing approach must solve the oracle problem [8] – that is, how to know whether a given test-case has been optimised, placed, and routed correctly? There is no single 'answer' to a place-and-route problem, so we cannot pre-determine what the output of the P&R engine should be (and to do so would be an enormous effort on our part anyway). We could simulate the pre- and post-P&R designs to check that they give the same results on a random selection of inputs, but this approach might miss bugs if the inputs are poorly chosen. We could do a formal equivalence check between the pre- and post-P&R designs, but this could be so slow that it ruins our testing throughput. We could check the pre- and post-P&R designs for a simple structural equivalence, but this would likely lead to false positives when structure-changing optimisations are applied during P&R.

6. Non-determinism. Some P&R engines are non-deterministic – their outputs may be affected by random seeds or small changes in the input. This could cause issues when trying to reproduce problematic test-cases. Moreover, combinational loops in the netlists themselves can cause the circuit to have non-deterministic behaviour; this may cause the P&R engine to error out and also makes equivalence-checking awkward, so should be avoided.

7. Secrecy. Commercial P&R engines do not expose their inner workings, so end-users can only test them as black boxes. In particular, we cannot use white-box fuzzing approaches like AFL [1], which rely on source-code coverage information to guide test-case generation. Then again, even with access to the source code, AFL is likely to be limited to finding crash bugs, which are less interesting than incorrect-output bugs.

1.2 Our contributions

In response to these challenges, we have designed FUZNET, a random generator of valid P&R netlists of tractable size, together with a procedure based on a combination of SAT solving and cycle-accurate simulation for determining whether the pre- and post-P&R netlists are equivalent, plus an efficient test-case reduction method that analyses symptoms of a bug to quickly produce a minimised design that still triggers the bug. FUZNET is a black-box fuzzer, meaning that it can be directly applied to commercial off-the-shelf P&R engines without requiring access to their source code.

We have used FUZNET to generate 57,316 random netlists for testing AMD's Vivado P&R engine, version 2024.2. About 3.3% of them failed pre- vs. post-P&R equivalence, and after test-case reduction, we isolated two functional bugs in logic optimisations carried out during P&R: in CFGLUT5 retargeting (confirmed and fixed by AMD) and in CARRY4 constant propagation (received by AMD but still pending confirmation).

Our work demonstrates that although the nonfunctional properties of P&R engines make them challenging targets for traditional fuzzing techniques, it is possible to design strategies to overcome these challenges, and thus obtain a valuable fuzzing technique that can identify subtle bugs in a mature P&R engine, and ultimately help to make them more trustworthy in safety-critical settings.

In summary, our main contributions are:

(1) The design, implementation, and evaluation of an open-source tool called FUZNET for generating random, valid netlists that can be used to test P&R engines.
(2) A testing campaign on a widely used commercial P&R engine (AMD Vivado 2024.2) and an analysis of the two logic optimisation bugs that this revealed.

[1] Of course, one might then argue that bugs found using netlists from outside the subset targeted by current synthesis tools are less important. Our counter-arguments would be: (1) some netlists are written or tweaked by hand, and they may not lie in this subset, (2) synthesis tools of the future may target a different subset, and (3) bugs revealed using netlists from outside this subset could also affect less "middle-of-the-road" netlists inside the subset.

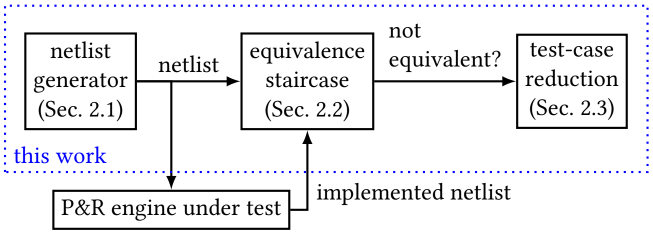

netlist generator (Sec. 2.1)
netlist
equivalence staircase (Sec. 2.2)
not equivalent?
test-case reduction (Sec. 2.3)
this work
P&R engine under test
implemented netlist

Figure 1: Our testing flow

Section 2 describes the design and implementation of FUZNET. Section 3 reports on how we evaluated its performance. Section 4 describes the P&R bugs found by FUZNET. Related work is discussed throughout the paper.

FUZNET is free and open-source software. It is archived on Zenodo [10] and developed on Github:
https://github.com/splogdes/fuznet

2 Our approach

FUZNET is based on the flow depicted in Figure 1, which we now explain in detail.

2.1 Generating netlists

In contrast to language-based fuzzers like Csmith [29] and Verismith [14], which work by building up the syntax tree of a program, FUZNET generates random hypergraph structures. (This is somewhat similar to the FuzzFlesh tool, which generates random control-flow graphs for testing decompilers [11].) The hypergraph generated by FUZNET represents a netlist, with each vertex of the graph representing a primitive and each hyperedge representing a net. The type of each primitive is selected from a dictionary of available primitives, which includes LUTs, arithmetic and logical operations, registers, and small LUT-based RAM blocks, but does not currently include DSP, BRAM, or URAM blocks.

FUZNET adds new primitives to the netlist one by one. The obvious way to do this would be to add a randomly chosen primitive to the netlist and have it driven by the output of an existing primitive. However, this would never introduce loops into the netlist.

So, FUZNET splits the netlist-extending process into three separate operations. `AddRandomModule` adds a new primitive to the netlist, attaching its inputs to existing nets and creating a new net for its output. `AddUndrivenNet` adds a new net to the netlist, leaving it undriven, while `DriveUndrivenNet` chooses an existing undriven net (say, X) and adds a new primitive (say, Y) to drive it. Crucially, Y might be placed so that it is also driven (perhaps indirectly) by X, thus creating a cycle. However, as mentioned in §1.1, we need to avoid *combinational* cycles. So, when FUZNET is choosing where to place Y, it avoids attaching it to nets that are already combinationally driven by X.

Our generation algorithm diverges from ordinary grammar-based generators regarding the size of the test-cases it produces. Typical generators produce test-cases whose sizes follow a geometric distribution (Fig. 2, left) because at each step of the generation

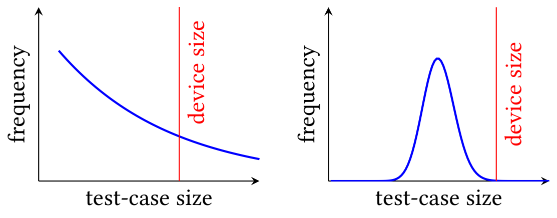


Figure 2: Distributions of test-case sizes

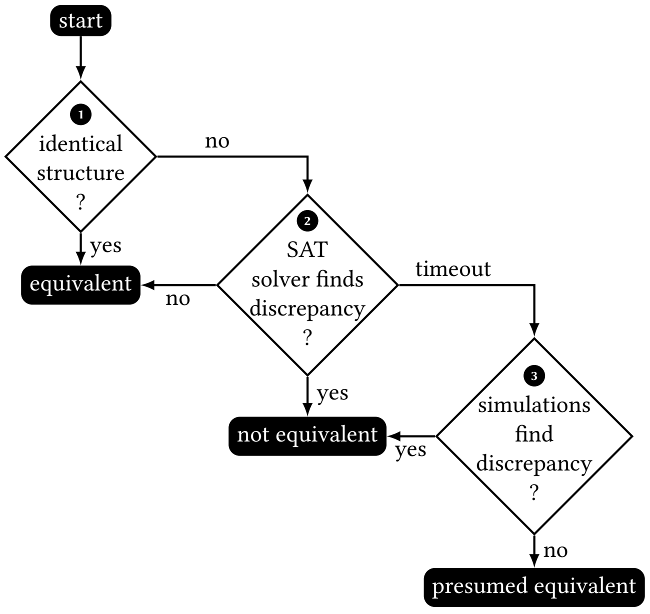


Figure 3: Our equivalence staircase

process, they stop with probability p and extend the test-case with probability $1 - p$. This is ill-suited to testing P&R engines because it generates many test-cases that are either too small, and hence not sufficiently challenging, or too large, and hence do not fit on the target device. Hence we generate netlists whose sizes follow a Poisson distribution, with its parameter set so that the distribution peaks when most of the target device's resources are used (Fig. 2, right).

2.2 Equivalence checking

We explained in §1.1 the challenges involved in checking pre-/post-P&R equivalence. Our solution is to build an 'equivalence staircase', that combines structural comparison, SAT solving, and simulation. As explained in §1.1, none of these techniques suffices *alone*, but it is possible to *combine* them effectively, based around the principle of succeeding or failing as quickly as possible. Our staircase is illustrated in Figure 3 and works as follows:

❶ We first compare the structure of the pre- and post-P&R netlists. If they have the same structure and use the same parameters and initial values, they can quickly be judged equivalent. This means that in cases where P&R has not changed the netlist's structure – perhaps because few optimisations were deemed necessary – we can get a quick resolution. If the netlists do not have the same structure, they may yet be functionally equivalent, so we proceed to the next stage.

❷ Our second stage is to ask a SAT solver to find inputs that make the pre- and post-P&R netlists produce different outputs. If the solver reports `SAT`, yielding such an input, then we have found a bug in the P&R engine. If the solver reports `UNSAT`, then we can be confident that the pre- and post-P&R netlists are equivalent. However, SAT solving can be very slow, so in the interests of high testing throughput, if a predetermined timeout is reached, we abandon SAT solving and proceed to the third and final stage.

❸ Our third stage is to use the Verilator cycle-accurate simulator on the pre- and post-P&R netlists with 100,000 random inputs. If we find a discrepancy between the pre- and post-P&R netlists, then we have found a bug in the P&R engine. If we do not, we deem the netlists likely equivalent, and move on to the next test case.

2.3 Test-case reduction

If we identify a pre-/post-P&R discrepancy, we reduce the test-case in order to obtain a small bug report for the vendors. Typical reduction algorithms slowly converge on a minimal program by chipping away at a bug-triggering program, gradually determining which parts can be removed or simplified while preserving the bug [21, 31]. This is feasible in domains such as compiler testing when re-running the system-under-test many times is cheap. But P&R is a slow process, so it is important that reduction converges quickly.

We can obtain a fast-converging reduction algorithm thanks to the following observation:

(1) When a pre-/post-P&R discrepancy is found, we are given a specific input vector that witnesses the discrepancy.
(2) Thus, we can isolate the specific output wires that disagree.
(3) Thus, we should be able to remove all the nets that are not in the 'fan-in cone' of those output wires (i.e. the nets from which there is no path to those output wires) while preserving the bug.

This yields a reduction process that can remove much of the netlist in a single step. In practice, however, we found that removing the seemingly-irrelevant nets would sometimes suppress the bug – this can be attributed to the unpredictable nature of bugs! And even when the bug is preserved, there usually remain opportunities to reduce the netlist further. So, our reducer follows up the coarse-grained 'cone reduction' step with a more fine-grained process that is more in-line with typical test-case reducers. It works by iteratively removing primitives from the netlist until a state is reached where any further removals would suppress the bug.

We described in §1.1 (challenge 4) a difficulty with test-case reduction in the context of P&R: if we remove part of a bug-triggering netlist and the bug goes away, we don't know whether this is because we removed the bug-triggering part of the test-case or because the smaller test-case simply required fewer optimisations to meet the given timing and resource constraints. FUZNET solves this challenge by arranging that whenever the reduction process claims to have removed the bug, it checks that all the original optimisations are still triggering, and if they are not, it calculates a new, higher target frequency based on the worst negative slack (WNS) and runs P&R again to try to restore the optimisations (and maybe the bug too).

2.4 Testing infrastructure

FUZNET incorporates several engineering features that help it achieve high testing throughput and convenient operation.

- All of the information required for a particular test run is captured in a seed. This seed, together with timing and status information about the run, is recorded in a CSV file. This allows any run to be easily reproduced.
- Whenever an error is detected, all artifacts from the current test run are captured, moved to a 'bug vault', and categorised by the type of error, ready for later inspection.
- It splits testing across multiple worker threads, coordinated by a top-level script, in order to maximise testing throughput.
- It uses a Nix development environment to ensure reproducible builds and identical runs across different machines.
- The overall testing framework is executed by `systemd`, which provides convenient features like restarting scripts after failures or reboots.

3 Evaluation

We now describe how we have sought to understand and improve the bug-finding ability of FUZNET. From an end-user perspective, the bugs that FUZNET can find in practice are likely to be of prime interest; these are presented in §4.

3.1 Preliminaries

Which metric to optimise. The first thing to establish is which metric our tool should be designed to optimise. The most obvious answer might be 'the rate at which bugs are found', but this is a poor metric for two reasons: (1) it depends on idiosyncracies of the particular P&R engine being tested, and (2) unless we have an effective way of automatically de-duplicating bug-triggering test-cases then we will likely just end up triggering the same bug repeatedly – that is: we would likely produce a large number of *test-cases that trigger a bug*, but a small number of *(unique) bugs*.

A somewhat better metric can be obtained after noting the fact that P&R engines typically record in their log which optimisations they have performed. If our testing campaign covers as many of these optimisations as possible, in as many combinations as possible, then it has a good chance of finding any bugs that exist. That is, we select as our metric: the *number of unique combinations of optimisations performed*. We say 'combinations' because given two optimisations, say `opt1` and `opt2`, we hypothesise that our chance of finding bugs improves if we have some test runs where only `opt1` fires, some where only `opt2` fires, some where both fire, and some where neither fire.

When expanding our testing campaign to open-source P&R engines in the future, we could also use more traditional code-coverage metrics here.

What parameters of FUZNET can we control? FUZNET incorporates various parameters, and having now established the desired metric to optimise, we can sweep through the values of these parameters to see which combination of values works best.

- `stop_iter_lambda`: This parameter specifies how many iterations of the generation loop are performed. It is linearly correlated with the final size of the netlist.
- `start_input_lambda`: This parameter specifies the number of inputs with which the netlist should be initialised.
- `start_undriven_lambda`: This specifies how many undriven nets are in the initial netlist.
- `prob_sequential_module`: This specifies the probability of adding a sequential (rather than combinational) module during netlist generation.
- `prob_sequential_port`: This specifies the probability that when a choice is made as to what drives a port, an element on which the port is sequentially dependent is chosen. The parameter is thus related to the probability of creating loops in the netlist.

Further parameters that could be tuned (in future work) include the timeouts of various subprocesses in the testing pipeline.

Questions to evaluate. We can now pose a list of research questions to evaluate.

RQ1 How many combinations of optimisations does testing with FUZNET cover, and how should FUZNET's parameters be tuned in order to maximise this metric? (§3.2)

RQ2 What testing rate can FUZNET achieve, and where are the bottlenecks that limit this? (§3.3)

RQ3 How effective is the 'equivalence staircase' we presented in §2.2 (§3.4)?

RQ4 How effective is the test-case reducer we presented in §2.3 (§3.5)?

Which P&R engines to test. As mentioned in the introduction, major commercial and open-source FPGA P&R engines include Altera Quartus Prime [2], AMD Vivado [4], nextpnr [23], and VPR [9]. We would have liked to have tested FUZNET on all four, but ended up only managing to test Vivado. VPR we have left for future work, while Quartus and nextpnr caused several difficulties.

Quartus was challenging to test for three reasons. First, several of the primitives are encrypted, which means we cannot use our SAT-based approach to compare the pre- and post-P&R netlists for equivalence (Fig. 3). Second, Quartus does not seem to support running P&R in isolation, without first running synthesis; this reduces our control over which netlists are used for testing. Third, Quartus seems to be stricter than the other P&R engines about which netlists it considers legal; for instance, primitives take a string parameter that specifies which input/output ports can legally be used, and our tool is not currently able to ensure that the netlists it generates match up with the values of all these parameters.

Nextpnr was challenging to test because it makes use of 'implicit connections', whereby I/O pads simply being in the right location means they are connected, despite this connectivity not being specified explicitly in the netlist. These implicit connections make it difficult to extract the pre- and post-P&R circuits for equivalence checking, as confirmed by the nextpnr developers [30]. It could

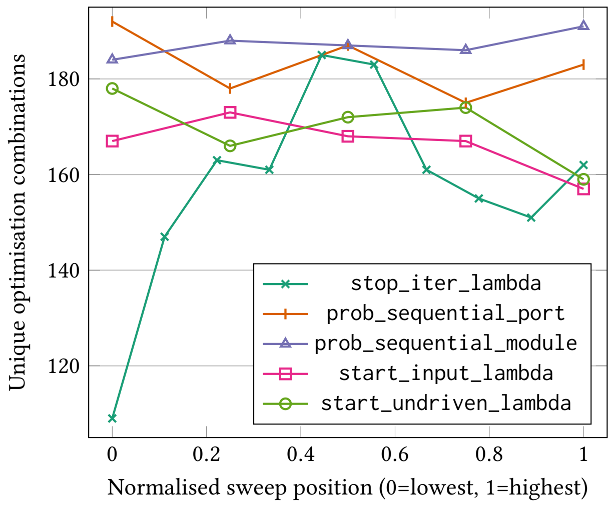
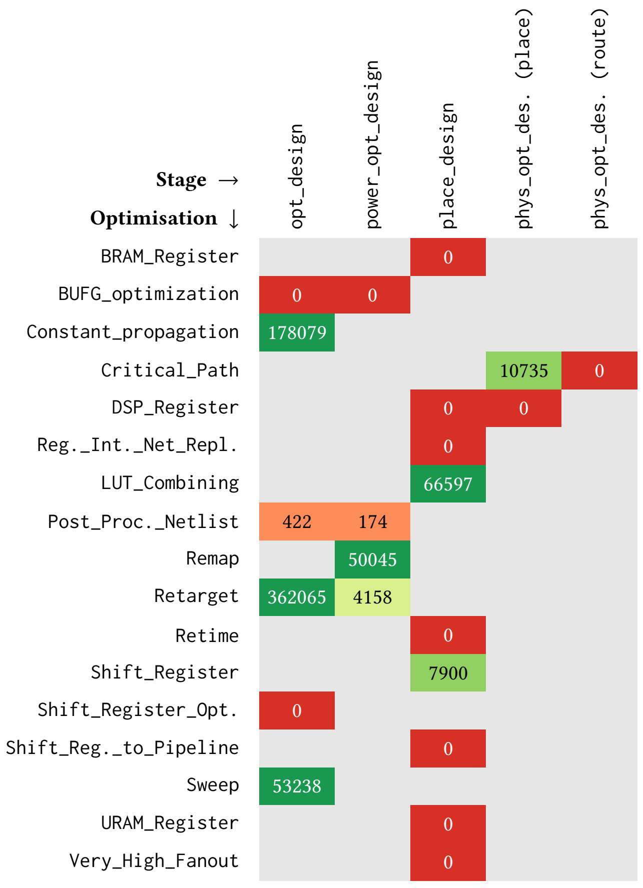


Figure 4: How our tunable parameters affect the number of unique optimisation combinations covered



Stage → / Optimisation ↓	opt_design	power_opt_design	place_design	phys_opt_des. (place)	phys_opt_des. (route)
BRAM_Register			0		
BUFG_optimization	0	0			
Constant_propagation	178079				
Critical_Path				10735	0
DSP_Register			0	0	
Reg._Int._Net_Repl.			0		
LUT_Combining			66597		
Post_Proc._Netlist	422	174			
Remap		50045			
Retarget	362065	4158			
Retime			0		
Shift_Register			7900		
Shift_Register_Opt.	0				
Shift_Reg._to_Pipeline			0		
Sweep	53238				
URAM_Register			0		
Very_High_Fanout			0		

Figure 5: A heatmap showing which optimisations in which stages our testing campaign covered. Grey: the optimisation does not run in this stage. Red–green: how many times, in total, the optimisation was applied.

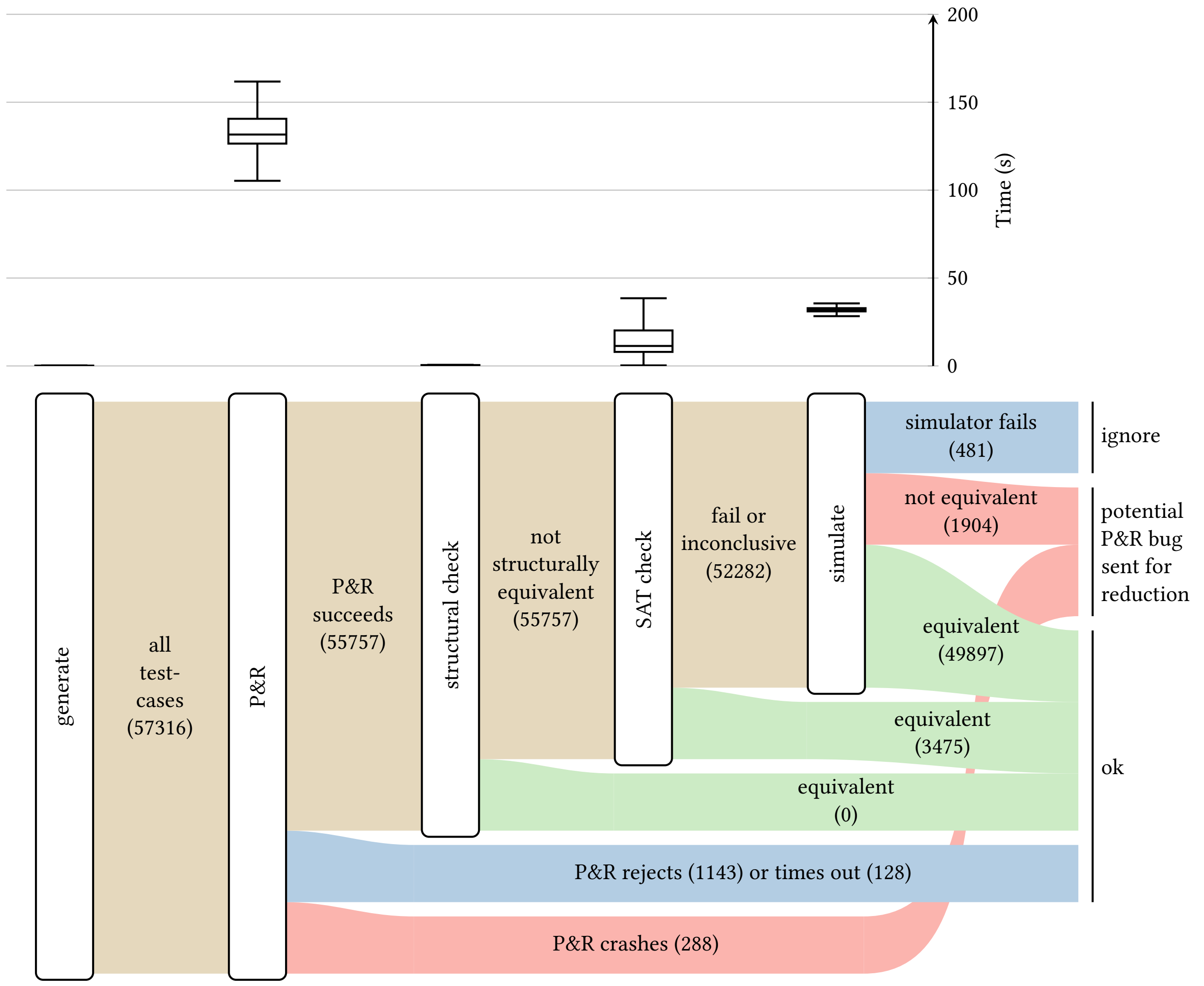


Figure 6: Top: Box plots showing the range of times taken by each stage of our testing flow. Bottom: an alluvial diagram showing the outcomes from the 57316 test-cases that we generated.

be possible to extend FUZNET with a 'tech mapping' that would provide this implicit connectivity information, but this would make FUZNET a lot more complicated – and likely a lot more buggy *itself!* We were also informed by the nextpnr developers that nextpnr does not actually carry out any netlist optimisations during P&R (unlike Quartus and Vivado), which makes it unlikely that our technique, which only compares the pre- and post-P&R netlists, would find any bugs. It is likely that VPR [9] would fall into this category too. (But FUZNET could become valuable for testing these tools if they incorporate logic-restructuring optimisations in the future.)

3.2 RQ1: Understanding and tuning FUZNET's optimisation coverage

Figure 4 shows how each of FUZNET's main tunable parameters (as defined in §3.1) influences the number of unique optimisation combinations. The range of each parameter is normalised to the [0..1] interval, so that the different parameters can be plotted together. For each parameter value, we ran FUZNET for 144 hours (12 threads and 12 hours per thread), and plotted how many unique optimisation combinations were covered.

We make the following observations:

- The `stop_iter_ lambda` parameter, which directly controls the number of primitives in the netlist, is the most obviously impactful. The number of unique optimisation combinations peaks near the middle of this parameter's range. This can be attributed to two distinct phenomena. When the value of `stop_iter_lambda` is to the *left* of this peak, the netlists are smaller, and hence there are fewer primitives to optimise. And when the value of `stop_iter_lambda` is to the *right* of this peak, the larger netlists take longer to pass through the P&R engine, which reduces testing throughput, and hence reduces the number of optimisation combinations covered in a given amount of time.

- As prob_sequential_port increases, we see a slight decrease in the number of unique optimisation combinations. This could be attributed to the fact that as prob_sequential_port increases, there is an increased likelihood of the netlist containing loops; these loops create more dependencies between primitives, which may reduce the room for optimisations.
- As prob_sequential_module increases, we see a slight increase in the number of unique optimisation combinations. This is possibly explained by the fact that some optimisations will only apply to sequences of nets.
- As start_input_lambda increases, we see a slight decrease in the number of unique optimisation combinations. This could be attributed to a netlist containing more inputs requiring more LUTs to capture its input space, and hence leaving fewer resources available for optimisations.

It is also worth assessing how extensive FUZNET's coverage of optimisations actually is. Figure 5 provides this assessment in the form of a heatmap. It shows the coverage obtained across all the sweeps described above. Each column corresponds to one of the five stages in Vivado P&R, and each row corresponds to an optimisation. A grey cell indicates that this optimisation is not applicable in this stage. In all the other cells, we record how many times the optimisation was covered, using a colour-coding between red (not covered at all) and green (covered many times). Our aim is for as many as possible of the non-grey cells in this table to be green.

We make three observations about our heatmap. First, we observe that most of the optimisations are carried out during the opt_design stage. These optimizations are particularly likely to fire on FUZNET-generated netlists because they have not already been optimised, unlike the netlists produced by synthesis tools. Standard optimisations like Constant_propagation and Sweep (which removes unused cells, legalizes connections, and optimizes macros) fire on almost every run.

Second, we note that the Retarget optimisation triggers the most often. This is likely because FUZNET generates netlists that contain primitives from several FPGA device families, and the Retarget optimisation is responsible for mapping these primitives onto those from the chosen family.

Third, we note that because FUZNET does not support DSP, BRAM, or URAM blocks, optimisations such as BRAM_Register are not triggered. It also generates netlists with only a single clock that is already buffered, so BUFG_optimization does not apply either.

3.3 RQ2: What testing rate can FUZNET achieve?

The boxplots at the top of Figure 6 show the range of times taken by each stage of our testing flow.

We make two observations. First, it is clear that the testing throughput is bottlenecked by the time taken to perform P&R. Our efforts to spread our P&R runs across multiple threads can thus be deemed worthwhile. FUZNET's average testing rate (single-threaded) is 19.7 designs per hour, with the average run taking 182.6s. Of this time, 74% is spent on P&R.

Second, the interquartile range for the 'simulate' stage is very small; in other words, Verilator's runtime does not change substantially depending on the test-case being simulated. Rather, most of its time is spent compiling the implementations of each primitive. Precompiling these would therefore be a sensible future optimisation for FUZNET.

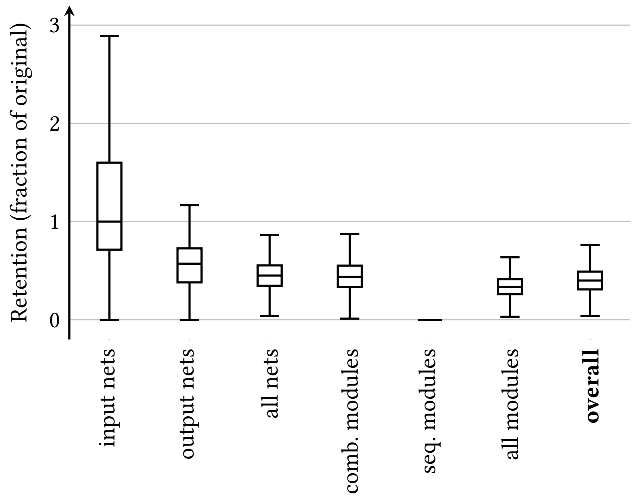


Figure 7: Effectiveness of our test-case reduction. Lower is better (more reduced).

3.4 RQ3: How effective is our 'equivalence staircase'?

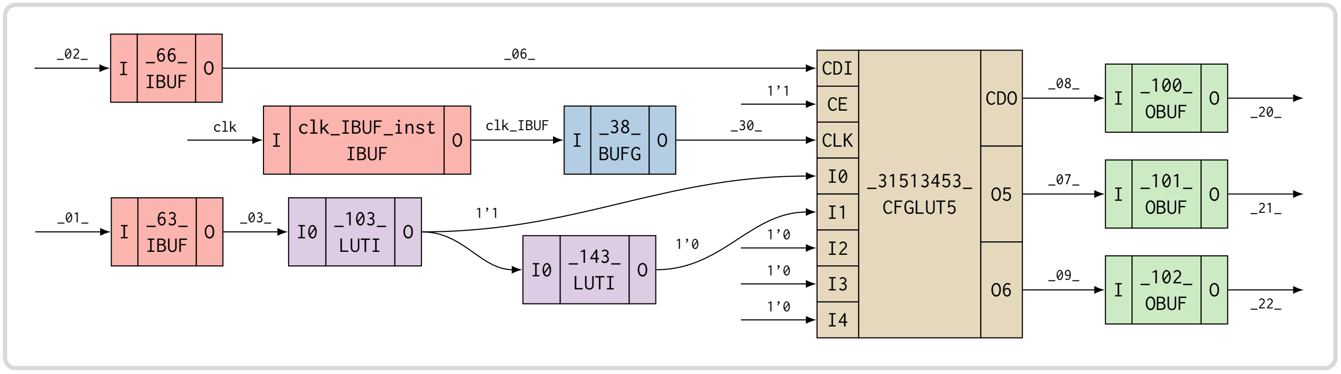
The alluvial diagram at the bottom of Figure 6 shows the outcomes from all the test-cases that we generated.

We make five observations. First, the structural check does not seem to be useful – none of the Vivado test-cases passed the structural check because in all cases, Vivado had modified the netlist's structure. However, the structural check takes very little time to run, so is likely worth leaving in. It was useful when doing initial testing on nextpnr, which tends *not* to modify the netlist's structure.

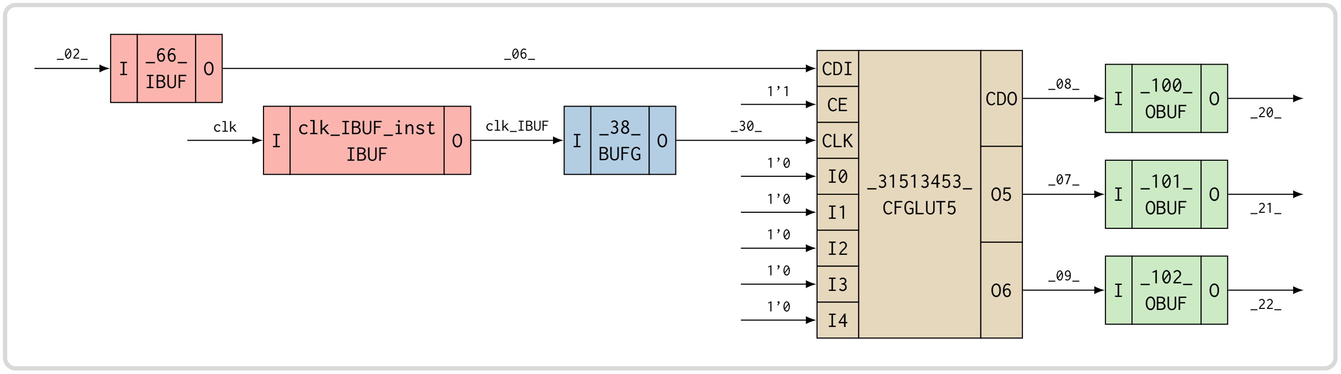
Second, the SAT check is able to verify pre-/post-P&R equivalence for about 7% of our test-cases, and thus avoid the need for these to go to the simulation stage.

Third, when the SAT stage identifies a discrepancy or times out, we run the 'simulate' stage. In theory, simulation is not needed once a discrepancy has been confirmed by the SAT solver, but we run it anyway as a double-check – this is a mild divergence from the principle described in Figure 3.

Fourth, all of the bugs we found were detected by the SAT solver (and then confirmed via simulation). In no cases did simulation reveal a bug after SAT solving timed out. In other words, when there is a discrepancy between the netlists, SAT solving tends to spot it quickly. This suggests that a possible optimisation to FUZNET would be to start a new testing run immediately after a SAT timeout, rather than running likely-futile simulations.



(a) Pre-P&R netlist



(b) Post-P&R netlist

Figure 8: CFGLUT5 retargeting bug

Fifth, some of the test-cases (top-right ribbon) fail simulation. Some of these are because FUZNET doesn't yet account for all the possible log messages and exit codes from simulation, and some are because of a suspected bug in Verilator where it crashes with an unhandled exception relating to indentation.

3.5 RQ4: How effective is our test-case reducer?

The boxplots in Figure 7 show the performance of our test-case reducer on 1730 test cases that fail equivalence-checking (both SAT and simulation). The size of the original netlist (i.e., before reduction) is at $y = 1$. Points below this line indicate that the netlist has become smaller, and points above it indicate that the netlist has grown. Each boxplot shows a different 'dimension' of the netlist that may have increased or decreased during the reduction process: the number of input nets, the number of output nets, the total number of nets, the number of combinational modules, the number of sequential modules, the total number of modules, and the overall size of the netlist (number of nets and modules combined).

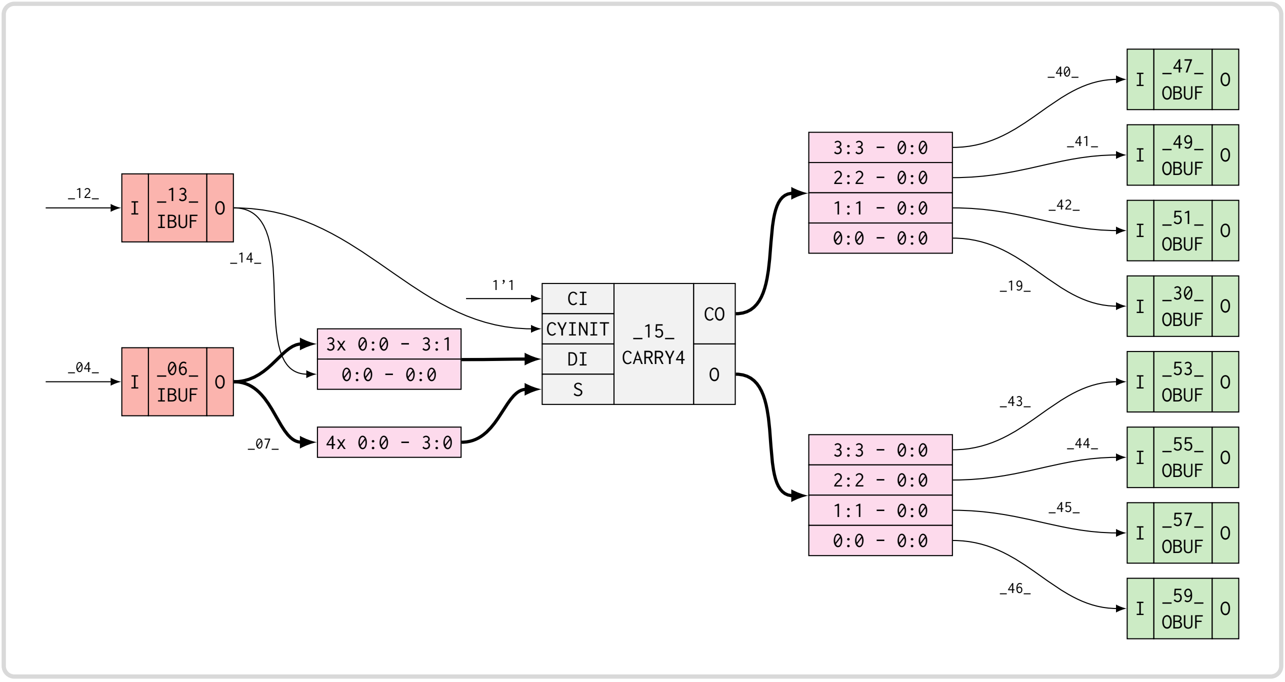
We make two observations. First, overall, we see from the rightmost boxplot that our reducer produces netlists that are on average about 40% of the size of the originally generated netlists. This is not a huge reduction factor when compared to those achieved in the context of testing conventional compilers, but it is important to bear in mind that FUZNET's netlists are not particularly large to begin with, in contrast to the output of typical C fuzzers like Csmith [29].

Second, the leftmost boxplot shows that our reducer often *increases* the number of input nets, and sometimes the number of output nets too. However, this is merely an artifact of more substantial elements being removed: our reducer is highly effective at removing sequential modules, and when it does so, it leaves behind many disconnected ports, all of which then count as additional input/output nets.

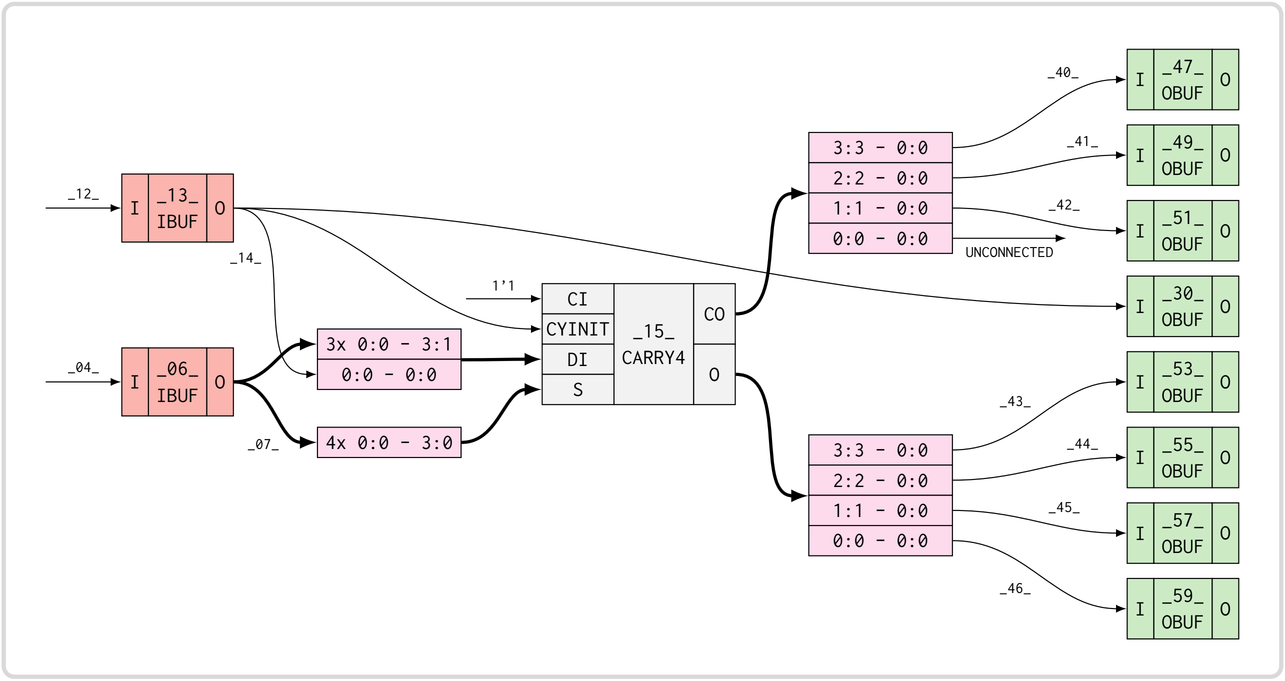
4 Bugs found

We now present two bugs that FUZNET revealed. Both the bugs were found in logic optimisations performed by Vivado 2024.2 during P&R. In both cases, we present the pre- and post-P&R netlists. The pre-P&R netlists were produced by FUZNET. In both cases, the bug was discovered using a larger netlist; what we present here are the netlists after test-case reduction has completed.

Each netlist is presented as a directed hypergraph of components. Components of the same kind are given the same colour to aid readability. Each component and each hyperedge is given a unique identifier such as `_101_`. Hyperedges with unconnected tails represent inputs, and hyperedges with unconnected heads represent outputs. `1'0` and `1'1` represent low and high constants.



(a) Pre-P&R netlist



(b) Post-P&R netlist

Figure 9: CARRY4 constant propagation

4.1 Bug: CFGLUT5 retargeting

Figure 8a shows a netlist generated by FUZNET. Vivado 2024.2 transforms that netlist into the one shown in Figure 8b. In Figure 8a, the CFGLUT5 component, which is a runtime-reconfigurable LUT [3], is taking a 1 on its `I0` input (produced by LUT `_103_`) and a 0 on its `I1` input (produced by LUT `_143_` which is behaving as an inverter). Vivado's `opt_design` pass has optimised this so that in Figure 8b, both inputs are constant 0s. One might expect that this discrepancy could be cancelled out by Vivado making the 'opposite' change to the functionality of the LUT; however, this particular LUT has its `CE` port held high, which means that its functionality is continually being reconfigured via the bitstream on `CDI` (produced by input `_02_`). Hence, regardless of the LUT's initial functionality, the two netlists are not equivalent. We were able to observe this during our equivalence check.

AMD confirmed the bug, filed a Change Request with the factory [6], and released a fix in Vivado 2025.2 [5]. As a workaround, AMD suggested explicitly annotating specific cells or signals in the RTL code with the `DONT_TOUCH` attribute, which instructs the tool not to optimise them [6].

4.2 Bug: CARRY4 constant propagation bug

Figure 9a shows a netlist generated by FUZNET that involves a CARRY4 block for performing fast-carry logic. Vivado 2024.2 transforms that netlist into the one shown in Figure 9b. In Figure 9a, output `_30_` is fed from one of the `CO` ports of the `CARRY4` block. Vivado's `opt_design` pass has used constant propagation incorrectly to deduce that output `_30_` can receive its value directly from input `_13_`. We know this because Vivado has annotated the block with `(* OPT_MODIFIED="PROPCONST" *)`. The two netlists are not equivalent, which we were able to observe during our equivalence check. An AMD staff member confirmed the receipt of our report and has passed it on to the tool developers [7]. Since then, we have asked AMD to confirm the bug [5], but have not received a reply at the time of writing.

5 Conclusion

We have presented a technique for randomised black-box testing of a critical component in the FPGA design flow: P&R engines. Our technique is implemented in an open-source tool called FUZNET, which we have used to generate 57,316 random netlists for testing AMD's Vivado P&R engine, leading to the discovery of two new functional bugs, one of which has been confirmed and fixed.

In the short term, there are various ways FUZNET could be extended in order to produce netlists that are more challenging for P&R engines, and hence more likely to expose bugs. One direction would be to add support for DSP, BRAM, and URAM blocks, and multiple clocks. Another would be for FUZNET to annotate a random subset of cells in the netlist with `DONT_TOUCH`, which would restrict the P&R engine's ability to use certain optimisations, and could coax it into doing fallback optimisations that may be less well tested.

In the medium term, a natural follow-on question is: can similar ideas be applied even *further* along the EDA flow; for instance, to FPGA bitstream generation? Doing so would be tricky because of difficulties understanding or observing the bitstream generator's output, but any bugs that *are* discovered could be very high-impact because bitstream generation occupies such a trusted role.

In the longer term, the results of our testing campaign could be used to justify the use of more rigorous engineering techniques when building P&R engines, such as programming them in a proof assistant such as Rocq and proving them bug-free. Such an approach has already been investigated for earlier stages of the EDA flow, including high-level synthesis [13, 15] and logic synthesis [18].

Acknowledgements

We are grateful to George Constantinides, Michalis Pardalos, and the anonymous reviewers of FPGA'26 for their help and valuable feedback, and to the EPSRC for their financial support via grant UKRI3221.

References

[1] 2025. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[2] Altera. 2025. Quartus Prime Design Software. https://bit.ly/quartusfitter.

[3] AMD. 2025. Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide, UG953. https://bit.ly/vivado-CFGLUT5.

[4] AMD. 2025. Vivado Design Suite User Guide: Implementation, UG904 (v2025.1). https://bit.ly/vivado_user_guide_impl.

[5] AMD Adaptive SoC and FPGA Support. 2025. Status of reported Vivado 2024.2 bugs. https://bit.ly/vivado-PnR-bug-update.

[6] AMD Adaptive SoC and FPGA Support. 2025. Vivado incorrectly optimizes away an always-high signal during opt_design. https://bit.ly/vivado-PnR-bug-CFGLUT5.

[7] AMD Adaptive SoC and FPGA Support. 2025. Vivado opt_design propconst incorrectly optimises CARRY4 logic. https://bit.ly/vivado-PnR-bug-CARRY4.

[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[9] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications, 7th International Workshop, FPL '97, London, UK, September 1-3, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1304)*, Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner (Eds.). Springer, 213–222. doi:10.1007/3-540-63465-7_226

[10] Oliver Cosgrove. 2025. *splogdes/fuznet: Fuznet 0.6.1.* doi:10.5281/zenodo.17702686

[11] Amber Gorzynski and Alastair F. Donaldson. 2025. FuzzFlesh: Randomised Testing of Decompilers via Control Flow Graph-Based Program Generation. In *39th European Conference on Object-Oriented Programming, ECOOP 2025, June 30 to July 2, 2025, Bergen, Norway (LIPIcs, Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:26. doi:10.4230/LIPICS.ECOOP.2025.13

[12] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. 2021. An Empirical Study of the Reliability of High-Level Synthesis Tools. In *29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2021, Orlando, FL, USA, May 9-12, 2021.* IEEE, 219–223. doi:10.1109/FCCM51124.2021.00034

[13] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. doi:10.1145/3485494

[14] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, Stephen Neuendorffer and Lesley Shannon (Eds.). ACM, 277–287. doi:10.1145/3373087.3375310

[15] Yann Herklotz and John Wickerson. 2024. Hyperblock Scheduling for Verified High-Level Synthesis. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1929–1953. doi:10.1145/3656455

[16] He Jiang, Peiyu Zou, Xiaochen Li, Zhide Zhou, Xu Zhao, Yi Zhang, and Shikai Guo. 2024. DeLoSo: Detecting Logic Synthesis Optimization Faults Based on Configuration Diversity. *ACM Trans. Des. Autom. Electron. Syst.* 30, 1, Article 12 (Dec. 2024), 26 pages. doi:10.1145/3701232

[17] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 65–76. doi:10.1145/2737924.2737986

[18] Andreas Lööw. 2021. Lutsig: a verified Verilog compiler for verified circuit development. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified*

Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 46–60. doi:10.1145/3437992.3439916

[19] NASA. 2013. Programmable Logic Devices (PLD) Handbook. https://go.nasa.gov/3Tf81Cf.

[20] Michalis Pardalos, Alastair F. Donaldson, Emiliano Morini, Laura Pozzi, and John Wickerson. 2024. Who checks the checkers? Automatically finding bugs in C-to-RTL formal equivalence checkers. https://dvcon-proceedings.org/document/who-checks-the-checkers-automatically-finding-bugs-in-c-to-rtl-formal-equivalence-checkers/.

[21] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346. doi:10.1145/2254064.2254104

[22] Raghul Saravanan, Sudipta Paria, Aritra Dasgupta, Venkat Nitin Patnala, Swarup Bhunia, and Sai Manoj P D. 2025. SynFuzz: Leveraging Fuzzing of Netlist to Detect Synthesis Bugs. arXiv:2504.18812 [cs.CR] https://arxiv.org/abs/2504.18812

[23] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. 2019. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*. IEEE, 1–4. doi:10.1109/FCCM.2019.00010

[24] Flavien Solt and Kaveh Razavi. 2025. Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 3941–3958. https://www.usenix.org/conference/usenixsecurity25/presentation/solt

[25] João Victor Amorim Vieira, Luiza de Melo Gomes, Rafael Sumitani, Raissa Maciel, Augusto Mafra, Mirlaine Crepalde, and Fernando Magno Quintão Pereira. 2025. Bottom-Up Generation of Verilog Designs for Testing EDA Tools. arXiv:2504.06295 [cs.AR] https://arxiv.org/abs/2504.06295

[26] Xilinx. 2017. Functional Safety Solution Brief. https://bit.ly/xilinx_functional_safety.

[27] Zhihao Xu, Shikai Guo, Xiaochen Li, Zun Wang, and He Jiang. 2025. SIMTAM: Generation Diversity Test Programs for FPGA Simulation Tools Testing Via Timing Area Mutation. *ACM Trans. Des. Autom. Electron. Syst.* 30, 2, Article 20 (Jan. 2025), 25 pages. doi:10.1145/3705730

[28] Zhihao Xu, Shikai Guo, Guilin Zhao, Peiyu Zou, Xiaochen Li, and He Jiang. 2025. A Novel HDL Code Generator for Effectively Testing FPGA Logic Synthesis Compilers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025), 1–1. doi:10.1109/TCAD.2025.3565488

[29] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. doi:10.1145/1993498.1993532

[30] YosysHQ/nextpnr. 2025. nextpnr-generic fails to connect IOBs for simple wire-through. https://github.com/YosysHQ/nextpnr/issues/1481.

[31] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. doi:10.1109/32.988498

[32] Yike Zhou, Yanyan Jiang, and Jian Lu. 2025. Unveiling Cross-checking Opportunities in Verilog Compilers. *ACM Trans. Des. Autom. Electron. Syst.* 30, 2, Article 32 (Feb. 2025), 23 pages. doi:10.1145/3715325

[33] Peiyu Zou, He Jiang, Xiaochen Li, Zhide Zhou, Shikai Guo, and Xu Zhao. 2025. Logic Synthesis Tools' Testing via Configuration Diversification With Combinatorial Multiarmed Bandit. *IEEE Transactions on Instrumentation and Measurement* 74 (2025), 1–17. doi:10.1109/TIM.2025.3542861