Creating a tool for FPGA Design Verification

Justin Hsu, Arsalan Ahmed, Daniel Narevich

Department of Electrical Engineering

University of California

Davis, U.S.A

Abstract - Field-Programmable Gate Arrays (FPGAs) are widely used in digital system design due to their reconfigurable nature and flexibility. However, ensuring the functionality of their internal logic, particularly Configurable Logic Blocks (CLBs), has always been a challenge. This project presents a software-based verification tool designed to detect stuck-at faults in CLBs by simulating logic behavior and generating targeted test vectors. The tool models five-valued logic and uses a custom PODEM inspired approach to evaluate fault propagation, along by a traditional implementation of the PODEM algorithm for comparison. Through automated test vector generation and simulation, the tool identifies faults based on output mismatches and enables precise fault localization. Designed with modularity and FPGA-specific structures in mind, the tool serves as a foundational step toward self-checking FPGA systems and facilitates further integration with hardware testbenches and simulation environments such as ModelSim.

Keywords: Configurable Logic Blocks, stuck at faults, fault propagation, testbenches.

I. Introduction

Digital design is pushing the physical bounds of layout and manufacturing. With modern chips with the transistor count in the billions, proper verification of the design is crucial in ensuring reliable, long-lasting use. With such complicated designs, ensuring correct circuit layout, and verifying logic outputs is of utmost importance in a valid design. Circuit faults such as Vdd being shorted to ground, incorrect logic circuits, and poorly designed modules can be caught early. By using verification methods to conduct automatic test generation to detect faults, circuit layout mistakes can be caught in the pre-silicon phase, thus saving money and time.

Furthermore, due to the unidealities in the manufacturing process, process variations are an inherent defect that must be dealt with. This could lead to incorrectly built transistors, transistors with the wrong sizes, or similar manufacturing defects that occur because of the complexities involved in the fabrication process. Additionally, external sources of contamination such as dust, particles, or other wafer imperfections can severely damage computer chips. As a result, verification must be performed on each chip to ensure proper functionality.

Stuck at faults is one of the most common errors that occur in digital design. Stuck at faults describe when a signal propagated by a wire, output, or input is stuck at a certain value permanently. This could be due to a variety of different factors such as incorrect circuit design, manufacturing errors and defects, or broken traces. As a result, test vectors must be applied to the circuit to determine whether these stuck at faults occur by propagating the currently being tested signal to the output. In our 16x1 mux alone, there were hundreds of possible stuck at faults. In modern digital design like processors, the number of possible stuck at faults could potentially range from millions to billions. While incredible tedious and time inefficient, we could have conceivably used pencil and paper to manually perform test generations by brute forcing test vectors-changing one value at a time. Clearly in a modern design with a countless number of potential stuck at faults, manual computation of

test vectors is not a viable solution for verifications both due to the sheer volume of vectors needed to be generated, as well as the time it takes. As a result, automatic test generation must be performed.

Automatic test generation can use a variety of different algorithms, but for our tests we employed fully random test generation, and two versions of the PODEM algorithm. Fully random test generation is very good at initially finding many test vectors. While random test generation is effectively the same as pencil and paper manual test generation, as both methods use brute force to find the vector, but random test generation is much faster. However random test generation eventually leads to diminishing returns-the more vectors already found the less new vectors will be found. This is because the remaining faults tend to require very specific input conditions to both activate the fault and propagate its effect to the primary output. Since random vectors do not account for the internal structure of the circuit or the nature of the faults, the likelihood of generating the precise input patterns needed for these difficult-to-detect faults becomes increasingly low. To address the diminishing returns of fully random test generation, algorithmic approaches are employed that base vector generation on the circuit's internal structure and logic behavior. As a result, the effectiveness of test vector generation increases significantly, particularly in detecting hard-to-reach faults. Our algorithm of choice was Path Oriented Decision Making-PODEM.

A. PODEM

1. **Fault Activation:** PODEM begins by identifying a target fault-for our case a specific stuck at fault line. Then PODEM assigns values to the inputs, so the output of the faulty circuit is different to the fault free circuit-i.e. fault activation
2. **Fault Propagation:** After activation, the algorithm ensures that the fault propagates to the primary ouput by carefully assigning values to primary signals such that the fault is propagated to the output.
3. **Backtracing:** PODEM uses a decision-driven backtracing process in which a primary input will be assigned a value that will propagate the fault. If this decision in primary input value leads to no fault propagation, PODEM backtracks and tries the alternate input value. This continues until a test vector is found or all possibilities are exhausted.

While for large circuits, the optimal way of leveraging automatic test generation is to first use random test generation to quickly find many initial test vectors and then switch to an algorithm to find the corner cases, our circuit was small enough that both random test generation and PODEM were able to generate test vectors in an appropriate amount of time independently.

This project applies PODEM to a 16x1 mux, which not only functionally verifies the mux as an independent unit, but also verifies the operation of an FPGA, as multiplexers serve as the backbone of the architecture of Look-up Tables (LUTs). [1] Muxes have the property of being able to implement any boolean function and can also serve as ways to create D flip-flops. The mux is an incredibly versatile component that also serves as the foundation of many datapath implementations for decision making in computer architecture. As a result, it is of utmost importance to ensure the correct operation of the mux, as so many systems and structures are tied to its use. While our system uses a large 16x1 mux, many systems use a smaller multiplexer for their lookup tables. We chose a 16x1 mux because we thought it would be large enough to not only showcase the power of the PODEM algorithm but also highlight the effectiveness in

verification of such an important component, and it was used in several research papers.

II. Objectives

The primary objective of this project is to develop an efficient, modular software tool for the verification of Configurable Logic Blocks (CLBs) in FPGA designs. The motivation stems from the critical need to ensure the reliability of the internal structures of FPGAs, which are often taken for granted during high-level verification processes. [2] Additionally, the verification of FPGA hardware is of utmost importance as FPGAs are able to implement any digital design without the need for designing the internal architecture, lending FPGAs to be powerful devices capable of deploying prototype digital designs in the pre-silicon phase, potentially saving millions of dollars in research and development, as well as months if not years of time.

Stuck-at faults in CLBs, if undetected, can lead to erroneous output behavior and system instability in deployed hardware. Our verification tool seeks to bridge this gap by providing a flexible and automated framework that not only 2 identifies faults but also supports their localization within the logic fabric. To achieve this, our tool models five-valued logic (0, 1, D, $\bar{D}$, and X) to enable precise simulation of fault effects and propagation through logic circuits.

The system supports multiple test generation methodologies. First, it includes a random test vector generator for baseline comparisons. Second, we implement PODEM (Path-Oriented Decision Making), a well-known ATPG (automated test pattern generation) algorithm in industry, as another test generation algorithm.

Our system is a 16x1 multiplexer that represents a Look Up Table in an arbitrary FPGA, that serves as a model to show the scalability of our test vector generator.

In even larger, more complicated designs than ours, random test vector generation plateaus as corner cases become harder and harder to randomly generate test vectors for and using PODEM takes a long time to generate the initial test vectors. The combination of random test vector generation and PODEM serves to be an effective method of not only drastically reducing the time to generate but also increasing the coverage of our generated test vectors.

Ultimately, our objective is to establish a groundwork for a future FPGA verification environment where faults can be detected and addressed, allowing for simpler verification of these products.

III. Methodology and Implementation

The tool comprises five main modules. The one module has controlled injection of stuck-at-0 and stuck-at-1 faults into selected locations within the logic block, such as specific muxes or logic gates. This module defines fault site parameters that are passed to the simulation engine. Another uses a five-valued logic system as mentioned earlier to evaluate the effect of the injected fault on logic behavior. The simulation models how the fault propagates through the digital circuit, providing insight into whether and where the fault goes to at the output.

In our simulator, each logic element is defined explicitly, this way we can inject stuck at fault errors into each line. Each logic block is defined with the corresponding boolean logic of the gate it represents (ie. AND, OR, NOR, etc.), and this way we can also quickly cycle through the various stuck at fault errors by using for loops to iterate through each line. This is an effective way of modeling these gates as we can systematically go through and simulate the stuck at fault errors. While this does increase time complexity to O(n*2^n) for each gate, due to redundant tests in the overall system we can expect the time to be faster than expected.

The actual structure of the 16x1 mux we implemented was then fully coded into python, utilizing the logic blocks we had defined previously to serve as the foundation. We know how to design a 2x1 mux using logic gates, calling it subdesign, and by cascading multiple 2x1 muxes together we were able to create the overall 16x1 mux, called design. While this harms scalability (as each new design would require the design to be synthesized from an HDL into explicit Python code), this method allows for the manual injection of stuck at fault errors as we can control each line within the overall design due to the granularity of our implementation. For additional reference, we also created an FPGA representation of the Look Up Table which could be used for our future work, such as using a functional FPGA for additional hardware verification. [3]

For test generation, we implement two algorithms for generating test vectors, particularly a random algorithm, which creates random input combinations for quick baseline testing. This method is a means of brute forcing test vectors, but as more test vectors are required for higher coverage, the success rate of generating random test vectors plateaus as it is highly unlikely to fully cover all corner cases within the system.

The other test generation algorithm used is PODEM (Path-Oriented Decision Making), which is an industry-standard algorithm used to generate vectors that propagate faults. PODEM is aware of the internal logic signals of the circuit, allowing it to make advanced predictions based on the distance of gates from primary inputs and outputs, as well as the gates themselves. By using backtracing, PODEM can deal with test vectors with don't-care values in them without needing redundant tests. This is because PODEM can directly tell which primary inputs will not contribute to fault propagation due to logic structure. The systematic approach PODEM takes to propagate the fault to the output and determine ambiguous test vectors is a large reason why PODEM is highly effective in fault coverage and fast.

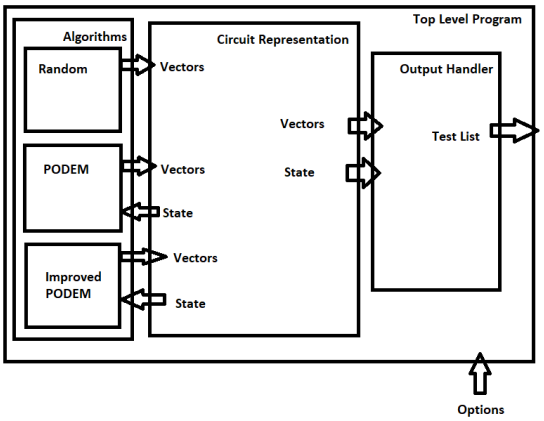

Fig. 1. Flow chart of our full test generator

We then compare outputs from the fault-free and fault-injected simulations. If a difference exists and can be attributed to the injected fault, the input vector is marked as successful for detection. All these results are then logged for both random test vector generation and PODEM, and a fault coverage test is run to determine what percentage of faults the generated test vectors from both tests cover. Furthermore, the total time it took to complete this generation is also reported.

IV. Results

Our three algorithms were all able to generate appropriate test lists for our Look Up Table architecture but performed very differently from each other. Before running our algorithm in Python, we performed a by-hand local fault collapsing for our circuit, which allowed us to test our algorithm on a minimized test list. From this paper analysis, we found 91.66% of our faults were non redundant, and there was a total of 45 test vectors required to create a local test list of our 16:1 MUX.

As we expected, the performance of the random algorithm decayed as it found additional test vectors but was still able to reach 91.66% full

non redundant coverage. To reach this coverage level consistently, which by our definition meant successfully generating a full test list for 5 iterations, took 320 random generations for each tested fault. We represent this value as “give_up” in our code.

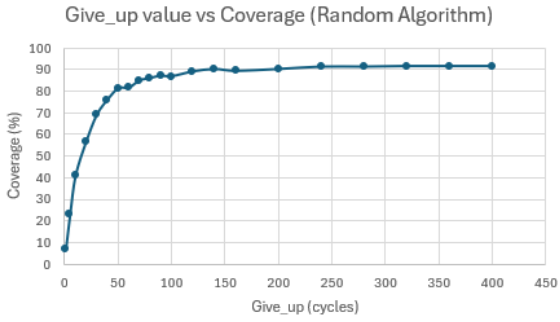

Fig. 2. Give_up variable value versus test coverage for the random algorithm

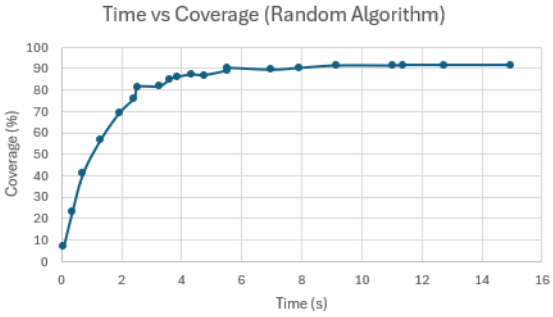

Fig. 3. Time for program to run versus coverage for the random algorithm

The random algorithm took just under 15 seconds to reach full coverage. It is clear however, that the efficiency of the random algorithm drops precipitously after around 3 seconds, or 80 cycles of operation. The overall speed of the random algorithm, which does not need to consult gates or know anything about the status of the circuit except for the primary output is at a clear advantage.

The next algorithm that we created was the PODEM algorithm. Unlike the random algorithm, PODEM is capable of eventually finding all non-redundant faults in the circuit. However, we found that our version of PODEM took a very long time to find test vectors. When told to generate a local fault list, our algorithm took 583 seconds to run, which is nearly 13 seconds per test vector. This time varied significantly based on what device the code was ran on and conditions, but our final time is based on operation on a personal laptop (LG Gram 2020) with Microsoft Excel being the only other program running at the same time. The reason for the increased operating time was likely due to our Python classes taking a very long time to respond to requests from the algorithm. Additionally, some test vectors require significant backtracing, increasing operation time.

Our final algorithm attempted to improve PODEM by reducing the number of operational calls to the circuit and design classes without compromising the accuracy of the algorithm. We do this by tracking which multiplexer our fault is contained within using the “faulty_mux” variable. This is then used in a case statement to pre-select selection lines without knowing any information about the circuit. This can reduce the operating speed by half for each selection line that we are able to preset as it reduces the number of possible test vectors for the PODEM algorithm to search through. The algorithm took 392s to find a full local fault list, or 9 seconds for each potential fault. This is a 46% speed improvement over the pure PODEM algorithm, which is a major improvement, however we see that not all selection lines could be preselected, especially when the fault is further from the primary inputs.

Overall, for our 16x1 multiplexer, it would be fastest to run the random algorithm exclusively to achieve full fault coverage, but we believe that a larger circuit would require use of our improved PODEM algorithm to achieve full fault coverage and generate the local fault list.

V. Conclusions and Future Work

To conclude the project, the Python-based tool we developed contained three algorithms.

Our results highlighted the strengths and trade-offs of each method. The Random method, while fastest, struggled to achieve full fault coverage, especially in complex circuits. PODEM, though which was more intensive and took longer, consistently produced optimized test vectors and showed its reliability for full verification. Our improved PODEM algorithm was a practical balance by significantly reducing runtime, operating close to half the execution time of the original PODEM algorithm, without compromising fault coverage accuracy. In a real-world design scenario, these algorithms would best be used in combination to achieve maximum fault coverage in the minimum time. We would begin by running the random algorithm to achieve a broad base fault coverage and then focus on key faults with our improved PODEM algorithm.

Looking forward to future iterations, our tool provides opportunities for more testing. We would like to expand it to support sequential logic as flip flops and RAM are key components making up a significant portion of an FPGA. [4] This would also make our program more functional in SRAM based FPGAs. [5] Since our Python program took a long time to run, with the gate class slowing down the two PODEM based algorithms, we would also like to optimize performance through using a faster language such as C or integrating our program with our HDL reference model. generalizing the architecture model for even more FPGA compatibility would also help our tool to truly benefit FPGA design as a whole. We would also like to integrate our project with self-checking and design for debug architecture which are increasingly in use in FPGA design. [6] With further development, including the exploration of machine learning and pattern generation, and structural optimizations, this tool has the potential to evolve into a scalable and adaptable verification platform suitable for modern FPGA design.

References

1. Hossein Mehri and B. Alizadeh, “An analytical dynamic and leakage power model for FPGAs,” pp. 300–305, May 2014, doi: https://doi.org/10.1109/iraniancee.2014.6999552.
2. J.-Y. Lee and E. Shragowitz, “Technology mapping for FPGAs with complex block architectures by fuzzy logic techniques,” Jan. 1995, doi: https://doi.org/10.1145/224818.224912.
3. Y. B. Liao, P. Li, A. W. Ruan, Y. W. Wang, and W. C. Li, “A HW/SW Co-Verification Technique for Field Programmable Gate Array (FPGA) Test,” IEEE Circuits and Systems International Conference on Testing and Diagnosis, pp. 1–4, Apr. 2009, doi: https://doi.org/10.1109/cas-ictd.2009.4960748.
4. W. Li, Y. Zhao, Y. Liu, and M. Chen, “SMEFF: A scalable memory extension fabric for FPGA,” Dec. 2017, doi: https://doi.org/10.1109/fpt.2017.8280119.
5. Chun-Lung Hsu and Ting-Hsuan Chen, “Built-in Self-Test Design for Fault Detection and Fault Diagnosis in SRAM-Based FPGA,” IEEE Transactions on Instrumentation and Measurement, vol. 58, no. 7, pp. 2300–2315, Jul. 2009, doi: https://doi.org/10.1109/tim.2009.2013921.
6. P. K. Lala and A. L. Burress, “Self-checking logic design for FPGA implementation,” IEEE Transactions on Instrumentation and Measurement, vol. 52, no. 5, pp. 1391–1398, Oct. 2003, doi: https://doi.org/10.1109/tim.2003.818545.

Author Biographies

Daniel Narevich is a student at the University of California, Davis. He is interested in digital systems and physical electronics and how they interact in real world devices.

Justin Hsu is a student of electrical engineering at the University of California, Davis. He is interested in VLSI and high-performance computer architecture.

Arsalan Ahmed is a student at the University of California, Davis majoring in Electrical Engineering. His interest is mostly around Digital Electronics and Circuit Fabrication.