A Flexible Resiliency Analysis Software Platform for Adiabatic Circuits

Jake Spracher, *Member, IEEE;* and Lee A. Belfore, II, *Member, IEEE,*

***Abstract*—Hardware fault injection is a popular method for evaluating the effect of random manufacturing defects on the performance of an electronic device or circuit. For some circuits, verification of correct operation can be accomplished by exhaustive testing of every possible input combination. For larger and more complex circuits, however, this is often not feasible. A 32 bit arithmetic logic unit for example would have $(2^{64}) * (total\ alu\ operations)$ input combinations, which would require years of testing even for very fast verification of test cases. A four operation 32-Bit ALU would require 47,978 years to verify using this method at rate of 20 nanoseconds per test case. For this reason, automated intelligent testing is critical to implementing a successful design. A custom software was developed in C++ that facilitates the testing and verification of adiabatic logic circuits by intelligent fault injection into gate-level SPICE netlist circuits.**

***Keywords*—Adiabatic Logic, Resiliency Analysis, Fault Injection, Automatic Test Pattern Generation.**

I. Introduction

Device reliability is an increasing challenge for implementation of next-generation, silicon-space efficient designs [1]. Because this is a recently emerging problem, new technologies often have few if any purpose built verification tools, which makes verifying their reliability difficult. Old Dominion University researcher Dr. Lee Belfore II and graduate Mihail Cutitaru are investigating a new single-phase partially adiabatic logic family (Input Decoupled Partially Adiabatic Logic) that has proven 67% more efficient than comparable adiabatic circuit implementations [2], but since adiabatic logic is a new technology, the infrastructure for design, development, and testing of such adiabatic systems is not yet available. A purpose built C++ program was developed that parses SPICE netlist circuit files into an organized structure and methodically injects simulated manufacturing defects, outputting a new netlist circuit file for each fault condition. The circuits described by these files can then be simulated using external simulation software to provide insight into the resiliency of the design to manufacturing defects as well as identifying features common to different fault conditions. The goal is to interpret this data and then develop test patterns to verify that fabricated adiabatic circuits are free of defects before they are analyzed.

II. Problem Definition

The goal was to devise a way to generate a suite of defective adiabatic circuits for simulation with realistic fault conditions. Fundamentally, the structure of the adiabatic circuit under test must be made available to the program. The design process for a digital system often begins with development of a hardware description language (HDL) representation of each device for initial testing and verification. HDL development environments often include features to convert the HDL description into a netlist or a hardware layout for fabrication, but do not offer good facilities for resiliency analysis or fault injection since these procedures must be done at the gate-level. The problem is compounded in the case of verifying adiabatic circuits since the gate level logic needs to be very precisely described. Consequently, L. Belfore developed a framework that allows the target HDL hardware descriptions to export identical descriptions in SPICE netlist format. The circuits in either representation needed to be somehow interpreted by the program and loaded into an organized structure. The program should reference a library of fault conditions and then decide a pattern by which to inject them into the circuit, injecting them one by one and exporting a new netlist for each fault condition. Researchers at The University of Michigan have developed an FPGA-based platform called Crash Test that is very similar to and more featured than the investigated software, but was not intended for adiabatic logic or interface with Belfore and Cutitaru's HDL Framework. A diagram of their software is shown below:

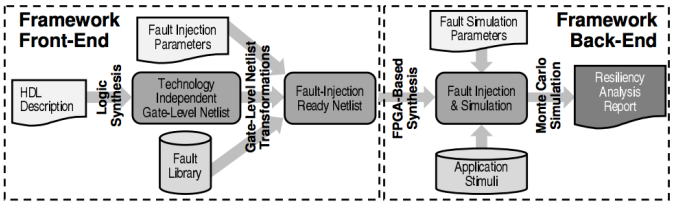

Fig. 1. An example resiliency analysis and verification procedure [3]

The investigated software functions very much like the "Front-End" interface of this diagram, where a gate level netlist is read from the HDL Framework after logic synthesis, fault injection parameters are given, and a fault injection ready netlist is generated for use in simulation. Simulation and resiliency analysis is currently handled externally but targeted for future development, since the unique constraints of adiabatic logic make the implementation of an automatic verification procedure very difficult.

III. Design Description

The software was purpose built to interface with specific HDL descriptions of adiabatic circuits, although it could

J. Spracher and L. Belfore are with the Department of Electrical and Computer Engineering, Old Dominion University, Norfolk, VA, 23508 USA e-mail: jakespracher@gmail.com, lbelfore@odu.edu.

be used to inject faults into any circuit given that it can be represented in netlist format. In its current version, the only requirement is that the netlist follow a regular naming convention, although functionality to interpret netlists that do not follow this particular convention could easily be developed. Since the software was purpose built, several methods of communicating the structure of the circuit between the HDL program and the fault injection software were discussed, although it was ultimately decided that no new method would be necessary and that the netlists themselves could be used to communicate the circuit structure, thus allowing for the aforementioned broad applicability of this software.

Specifically, the software in its current version is built to be run via command line. Calling the program with no command line arguments outputs information about its usage, including various options that specify the type of fault or faults to be injected, the fault injection pattern, desired modification to the netlist models and simulation parameters, and the desired resolution for fault conditions that generate netlists with incorrect syntax. Additionally, detailed documentation can be found in the appendix section.

A. Data Structures

Gate level netlist circuit representations can range from tens to thousands of gates that can have interconnections on an exponential order. For this reason, the data structure used to represent the circuit must be extremely efficient. The representation of the circuit resulting in the lowest space complexity is constructed by simply storing the netlist itself accessible by line. This is because a netlist maintains information about the connections to each component and each net simultaneously, which saves a lot of space. Unfortunately, the time complexity for extracting most information about the structure of the circuit, particularly, the components that connect to a given component, is very poor. In order to determine the connections to a given net, the entire netlist must be iterated from start to finish, which is very time-expensive for a netlist with thousands of lines. Accordingly, a better data structure needed to be developed.

A representation of the circuit that allows most information about circuit complexity to be accessed very quickly was created and used in this software, called the Circuit_Graph. The Circuit_Graph stores each component in a hash table. Each component maintains its type as well as a doubly-linked list of its terminals, and each terminal maintains the name of the component it resides on, the net that it is connected to, and a doubly-linked list of the terminals that it is connected to. Subsequently, the Circuit_Graph maintains all information about the circuit it is connected to in the absence of using nets to manage connections. Unfortunately, this data structure is extremely space-expensive, and the time complexity of populating it or altering a component is O(n^2) in the worst case where each component connects to every other component. In this worst case, the space complexity is also O(n^2) since each component will maintain a list of n connections. While this exact scenario is impractical, as the inter-connectivity of a circuit of fixed size increases, the time and space complexity also increases. Thus, for circuits with moderate to high inter-connectivity, the time and space complexities are worse than O(n). For this reason, the circuit graph is not the only data structure utilized, and it is also not fully populated in the investigated software.

In its current revision, the C++ program parses the netlist line by line and stores each component in the Circuit_Graph hash table by its netlist identifier. The program recognizes the type of each component quickly by capitalizing on the regular component naming patterns used by the HDL program, and stores it accordingly. An additional data structure is also maintained the stores the connections to each net. The nets that connect to each component are maintained in the Circuit_Graph of components, and a new, separate array of hash tables is maintained to manage the nets. The array index is the net number, and the hash table associated with each net hashes the identifiers of each component connected and stores a ”connection” data type. Each connection maintains its name, a linked list of the terminals on its component that connects to this net, and another linked list of any terminals that were connected to this net for the purpose of injecting a fault that needs to be later removed. This is discussed in more detail in the algorithms section. In this new structure that combines the Circuit_Graph and the array of nets, the Circuit_Graph does not maintain the inter-connectivity of every component. The lists of connections at each terminal are empty by default, and are only populated by the fault injection algorithms in circumstances where utilizing them could be faster than utilizing the array of nets alone. Accordingly, the space complexity of the software as a whole and the time complexity of most operations is O(n) in the worst case. For smaller circuits this is adequate, but further optimizations should be made, since generating a suite of test cases for a large circuit takes a very long time.

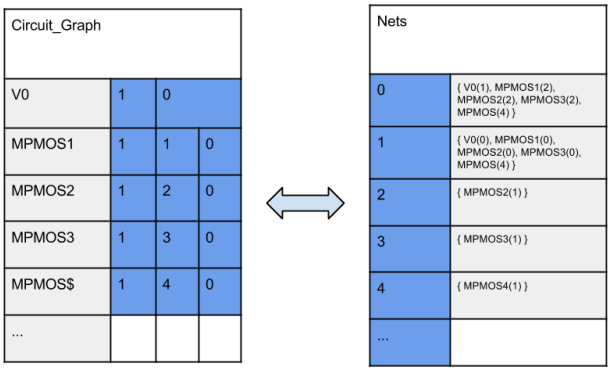

Fig. 2. Simplified representation of revised Circuit_Graph data structure

B. Algorithms

While the algorithm for parsing the netlist into the data structure is fairly straightforward. Many of the other algorithms

are relatively complicated. The most fundamental algorithm prints a netlist with the configuration contained within the data structure and the desired simulation information. The algorithm is as follows:

"Print Netlist" Algorithm Outline

1) Iterate the Circuit_Graph
   a) Output the identifier
   b) Iterate the component's terminals
      i) Iterate nets and find this terminal on this component
      ii) Output the net it was on
   c) Output the desired simulation parameters
2) Output the component models
3) Output ".END"

The most challenging algorithms to develop pertained to fault injection. Because of this, the software is currently only capable of inserting full bridge faults, although stuck-at and stuck-open faults are targeted for future development. To insert a bridge fault with minimal modification to the original circuit, the algorithm selects the first terminal of the component to be shorted as a focal point, and connects every terminal on every component that is connected to the subsequent terminals to this focal point. A highly simplified description of the algorithm to generate a suite of full bridge fault scenarios is as follows:

"Generate Full-Bridge Faults" Algorithm Outline

1) Allocate structures to save connections added to the focus and the nets originally connected to each terminal
2) Iterate the Circuit_Graph
   a) Rename the outfile
   b) Clear saved information from last fault
   c) Re-size the structure that stores connections added to focus so that it has as many lists of connections as component terminals
   d) Iterate all of the terminals on the component being shorted, skipping the focus
      i) Add all of the connections from the other terminals to the focus
         A) Add all of the connections added to the structure of connections added
         B) Add all of the connections to the net the focus is connected to
      ii) Clear the nets the non-focus terminals are connected to
      iii) Set the nets of the non-focus terminals equal to the net of the focus terminal
   e) Call a version of Print Netlist that prints all of the components in the Circuit_Graph EXCEPT the one being shorted
   f) Remove the fault
      i) Iterate the list of nets the removed components terminals were originally connected to
         A) On the focus terminal, remove all of the added connections
         B) On subsequent terminals,
         C) Set this terminals net back to its original, from the saved list of original terminals
         D) Put all of the connections back on the original terminals net

With some effort, this algorithm could be significantly improved by targeting the terminal that has the most connections as the focus of the short. In the case where one terminal has significantly more connections than the others, that terminal is a more desirable choice because only connections to the non-focus point terminals need to be made by the algorithm. Connections to the focus already exist.

IV. Results

The software was used to generate a suite of fault scenarios for an adiabatic buffer, and an adiabatic implementation of a Kogge-Stone type fast adder. The Kogge-Stone adder is a very large circuit, containing over 3000 components, and was tested for the purpose of evaluating time efficiency. The Kogge-Stone Adder (KSA) was loaded into the data structure in 7 seconds, and its unmodified netlist was output in another 8 seconds. Subsequently, each fault scenario took about 20 seconds to output.

The buffer, being a much smaller circuit, was processed by the software almost instantaneously. Select waveforms from simulation of the unmodified buffer circuit and a buffer with a single bridge fault are provided below. Plotted for each simulation are the power clock (red), one of the X inputs (cyan), one of the Y inputs (Pink), the X output (Green), and the Y output (Blue). To verify correct operation of the buffer, the X input and Y input should match the behaviour of the X output and Y output. One of the distinguishing features of Belfore and Cutitaru's novel adiabatic logic family is the sinusoidal power clock, which allows for power consumption to be reduced, and accounts for the sinusoidal behavior on the output waveform that is not present on the input.

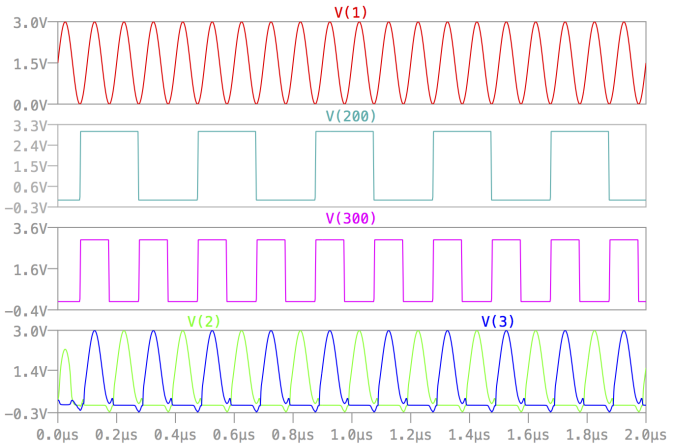

Fig. 3. Test waveform for an adiabatic buffer. Waveforms plotted in order: power clock (red), X1 input (cyan), Y1 inputs (Pink), X output (Green), Y output (Blue)

In the fault scenario simulation, it can be observed that the waveform titles do not match those from the original. This is because some of the signals have been shorted together as a result of the bridge fault, and consequently, some signals no longer exist since they have been merged with others. This anomaly requires manual selection of signals for verification, which can be complex and time consuming. A targeted improvement is the automated addition of fault analysis logic to the netlist. The fault injection algorithm would keep track of which signals have been merged or deleted, and map the appropriate new signals to additional components for probing in simulation. It can be observed that the X and Y outputs

now both exactly match the power clock, indicating that they have been shorted into direct contact with this signal. This is valuable information that indicates a possible behavior that could be tested for.

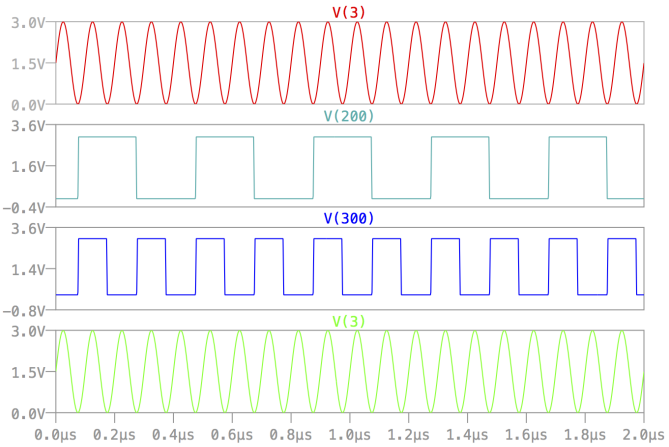

Fig. 4. Test waveform for an adiabatic buffer with a full bridge fault on a single transistor. It can be observed that the output signals are shorted to the power clock.

V. Conclusion

While several improvements are target for the future, a basic, custom built software fault injection platform was successfully developed for use with adiabatic logic circuits. Addition of an intelligent algorithm that decides where faults would be most effectively injected specifically for adiabatic circuits would be a very valuable research contribution, possibly worth of publication. Until such a platform is developed, a version of this software will be made available online for free use and open source development.

Appendix A
Documentation

A. Calling The Program

The program is executed by command line. Calling the program with no options will print a description of usage and options.

B. Execution Options

The program can easily be called with a single argument specifying the input netlist. This will generate a suite of output circuit files describing every possible fault scenario the software can generate. For large circuits this may not be desirable. The second and subsequent arguments of the program specify options. The program can be called with the option -at followed by the identifier of the component where a fault is desired and a single output file with a bridge fault on that component will be output.

C. Netlist Input Restrictions

The input netlist currently only supports sources, P-type MOSFET transistors, and N-type MOSFET transistors. It is required that the identifier of a source begin with ”V” or ”v”, the identifier of a PMOS transistor begin with ”Mp” or ”MP”, and the identifier of an NMOS transistor begin with ”MN” or ”Mn”. Currently, any simulation information, parameters, or models will be output in the fault ready netlist with no modification.

References

[1] S. Borkar et al. ”Design and reliability challenges in nanometer technologies”, In DAC-41, 2004.

[2] M. Cutitaru, L. Belfore II, ”New Single Phase Partially Adiabatic Logic Family”, *Proc. of Intl Conference on Computer Design*, pp. 9-14, 2012.

[3] M. Pellegrini et. al., ”CrashTest: A Fast High-Fidelity FBGA-Based Resiliency Analysis Framework”, 2010.