Simulation of Multi-input gates with Distinguishing Xs and X-Trees

Ryan D. Burrow

ECE 5505 Final Report

Keywords: Testing and Verification, Logic Simulation, Distinguishing X's

Contents

List of Figures

List of Acronyms

ATPG Automatic Test Pattern Generation

UI User Interface

Chapter 1: Introduction

1.1 Overview

The goal of this project was to transform the previous project, which implemented a simulator of circuits containing distinguishing Xs, but only on dual input gates, for gates containing more than 2 inputs; the process of implementing this and the challenges of doing so will be discussed in chapter 2. Another goal of this project was to implement some features that would help gain information from X values that were not squashed and were propagated to the output. The mechanism that was chosen to do this was named an X-Tree, and is a list of all the distinguishing Xs that make up an X value at any gate in the circuit. In addition, the first occurrence of each distinguishing X value and its inverse is tracked in two separate lists, to help identify key gates in the propagation path; this is covered in chapter 3. The final step to help obtain more information about these values was to implement an interface that would allow a user to interact with the circuit and inspect specific gates, as well as manipulate their output to see the effect on the circuit; this will be discussed more in chapter 4. A repository containing all of the code for this project, as well as a README and the LaTeXsource for this document can be found on github[1].

1.2 Background Research

While not much research was needed to implement multiple input gates for distinguishing X's, a notable amount was indeed needed before working further on the non-squashed X's. One of the primary papers I found was [1], which outlined two methods for dealing with

[1] https://github.com/RSAkidinUSA/ECE5505-Final-Project

X's in logic testing. One method was to implement distinguishing X's like had already been done in a previous project for this class, while another was to utilize a method of "Flip[ing] Fanout Bits" which entailed setting all of the inputs of a gate under test to X except one, flipping the non-X input, and seeing what changes occurred at the output. This method was then repeated for all inputs of gates of interest. The final method of the paper was to combine these two methodologies into one test. The idea of bit flipping was incorporated into the implementation of the User Interface (UI) and the idea of X-Trees was an expansion of the distinguishing X's method. Several other papers were investigated for more ideas on the topic. In [2] a different approach to utilizing the unique X values was investigated in which the values were used to model the "unknown behavior of a logically bound defect", which is something that is somewhat covered by the X-Tree model. In [3] a method for reducing search time for the controlling candidate of a fault was investigated, but as my implementation did not automate this process, this was not of great use to me. In [4] a further investigation into the effects of distinguishing X's and their relation to XOR gates was covered, which is something that took several revisions of code to accurately model, and is discussed in section 2.3. Finally, [5] covered a method of utilizing Boolean formulas alongside distinguishing X's which was interesting, but not of much use or relevance to my project. In most of these papers, the information gained from these distinguished X's, especially determining important gates, is used in developing higher output Automatic Test Pattern Generation (ATPG).

Chapter 2: Multi-input gates

2.1 Overview

The first step in completing this simulator was converting the code from the previous project to work with gates containing 3 or more inputs, rather than just 2. For AND gates and OR gates this was a relatively small change, however XOR gates presented several more challenges, some of which were only solved through the use of X-Trees, which will be covered in section 3.2.

2.2 AND/OR

There were two major things to consider in an AND/OR gate. The first was whether a controlling value was found; if this was the case, the simulator could stop checking other values and immediately pass on the controlling value to its successors. This was how it was implemented in the 2-input version of the code, so no change was needed. The second thing to consider was the implication of multiple X's. In the two input version, if two X's with different values were encountered, the simulator would stop checking inputs and pass on a new X value to the successors. This is not the appropriate behavior for multi-input gates though, as if a controlling value is also an input after these 2 unique X values, like shown in figs. 2.1d and 2.2d, it would be skipped, so this needed to be fixed. In addition, the order of inputs could matter, as if two unique X values were encountered and the third was the inverse of one of the previous, this should force a controlling value, however, in implementation it would output a new X value. An example of this would be figs. 2.1b and 2.2b if inputs 1 were directed to input 3 instead, and the new input 1 were a unique X value, X2. This was

solved partially through the use of X-Trees.

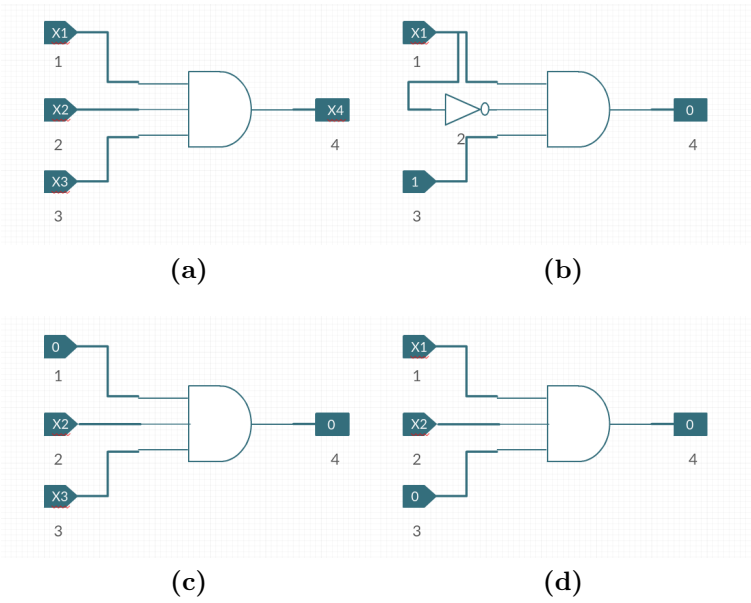

Figure 2.1: Various Multi-input AND Gate Configurations

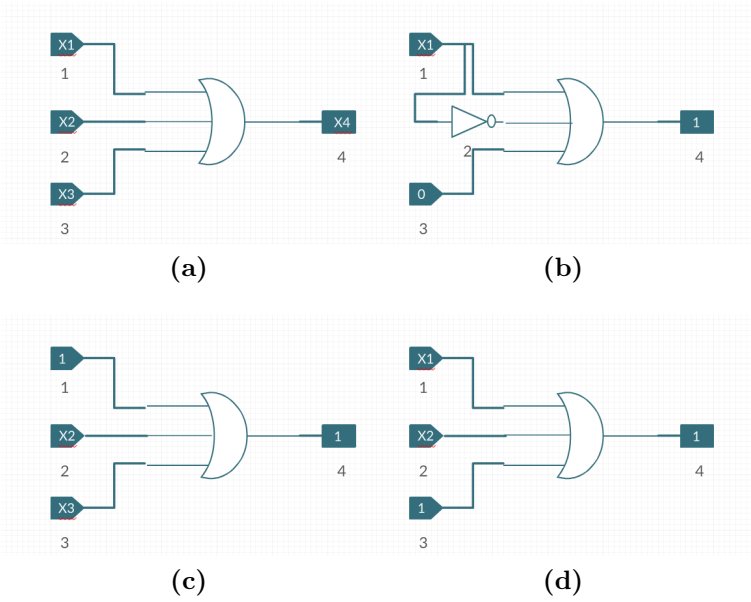

Figure 2.2: Various Multi-input OR Gate Configurations

2.3 XOR

XOR gates came with their own complications when converting to multi-input as opposed to dual input. For starters, XOR gates do not have a controlling value that allows the simulator to simply pass on that value, so all inputs must always be checked. Furthermore, using the implementation that was created for 2-input gates, the order of the inputs would matter. This is because after two unique X's were encountered, such as in fig. 2.3d, the system would always create a new X value. To counter this, the simulator needed to keep track of all previous X values encountered, to see if they could be canceled, or to see if they should evaluate to 1, like if an X value and its inverse were encountered. This was actually solved through the use of X-Trees, which tracked X values and could be used to restore an X value if all other competing values were eliminated.

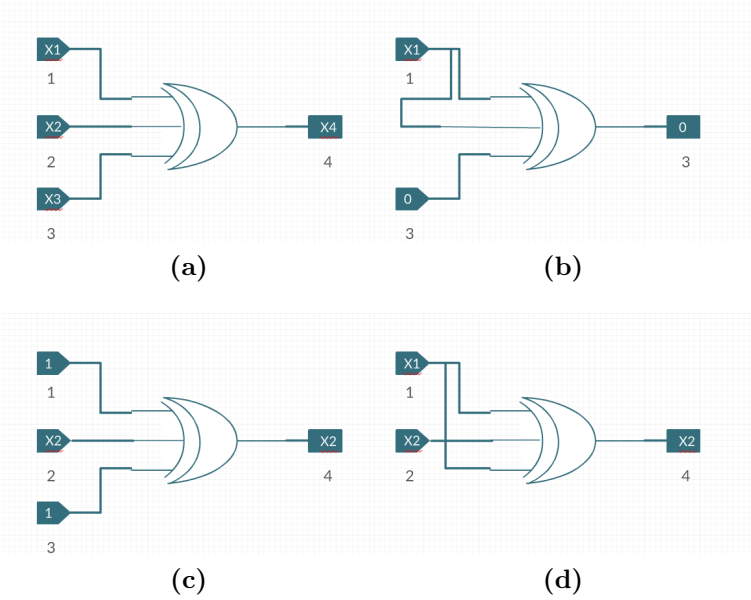

Figure 2.3: Various Multi-input XOR Gate Configurations

Chapter 3: X-Trees

3.1 Overview

The X-Tree was a structure that was created to help the user track how X values mixed and evolved through a circuit. It is simply an array for each gate with a value corresponding to each X value. X_TREE_NA (0) represents that the current X value is not in the history of X values for the current gate. X_TREE_ON (1) represents that the current X value is present in the history of the current gate. X_TREE_OFF (2) represents that the inverse of the current X value is in the history of the current gate. These values were chosen because X_TREE_ON and X_TREE_OFF are not mutually exclusive, and in an XOR gate or an OR gate would evaluate to 1, while in an AND gate would evaluate to zero.

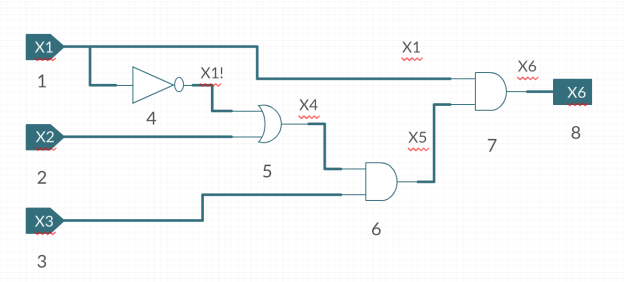

Figure 3.1: The circuit used in the example simulation

```
root@HAL9000:~/ECE5505-Final-Project# ./circuit-simulator-burrow.app -xo tests/xtree-test
Successfully read in circuit:
        3 PIs.
        1 POs.
        0 Dffs.
        9 total number of gates
        6 levels in the circuit.
vector #0:
        X

Gate 8's XTree:
        X1      at gate 1
        X1!     at gate 4
        X2      at gate 2
        X3      at gate 3
        X4      at gate 5
        X5      at gate 6
        X6      at gate 7

Number of vectors: 1
Number of clock cycles elapsed: 0
```

Figure 3.2: An example of the X-Tree output of the simulator

3.2 Effects on Gates

3.2.1 AND/OR

The use of X-Trees allowed for the resolution one of the problems referenced in section 2.2; the problem of order of inputs. Because all of the X value inputs are now stored, by a simple check of whether the X_TREE_ON was present as well as the X_TREE_OFF value, the simulator can quickly check if both an X value and its inverse are inputs to the gate, and if so it can quickly be determined that the controlling value for the gate must occur and an output can be forced.

3.2.2 XOR

The use of X-Trees also allowed for a similar resolution in XOR gates, in which case the X value of each input was not simply added to the X-Tree of the new gate, but rather XOR'd. This allowed repeated X value inputs to be removed, and allowed the simulator to quickly and easily check if newer inputs had canceled out an older X value, de-escalating the X value if necessary.

Chapter 4: User Interface

The UI was created with the intention of allowing a user to get a better understanding of the circuit through various means. The first option given to the user is the ability to view X-Trees at any gate, which allows them to track how X values change. These X-Trees include not only the X values in the tree, but also the first occurrence of each value, useful for determining where values join and evolve. A sample output of this use case can be seen in fig. 4.1. The second option that was included in the UI was the option for a user to modify the output of a gate, and see how the tree is affected. An example of this can be seen in fig. 4.2, in which driving a gate to value 0 forces the output to zero. This can be useful in tracing which gates are relevant or controlling in a circuit, and can be helpful in quick modifications of circuit input to try and gain ideal values. The gate control option could be evolved into an automated task for ATPG; however, I decided that this was outside of the scope of my project and what I would be able to complete in the time given.

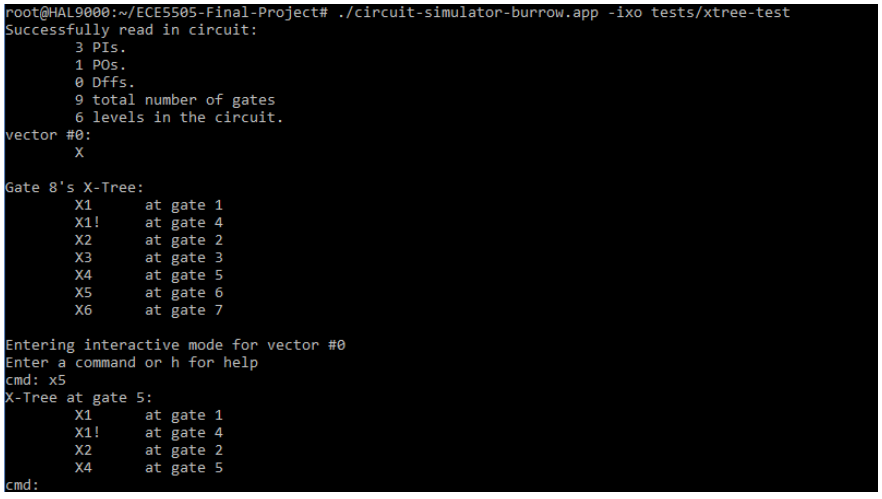

Figure 4.1: An example of the user interface and how to observe the X-Tree of any gate in the circuit

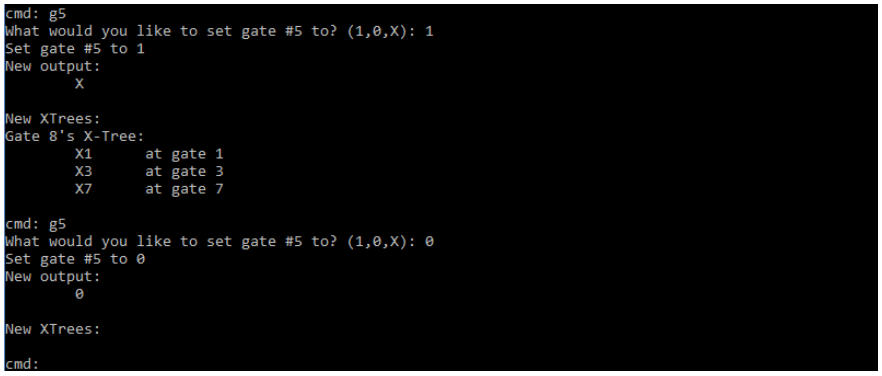

Figure 4.2: An example of the user interface and how to force a value on the output of a gate

Bibliography

[1] N. Sridhar and M. S. Hsiao, "On efficient error diagnosis of digital circuits," in *Proceedings International Test Conference 2001 (Cat. No.01CH37260)*, Nov 2001, pp. 678–687.

[2] X. Yu and R. D. Blanton, "Multiple defect diagnosis using no assumptions on failing pattern characteristics," in *2008 45th ACM/IEEE Design Automation Conference*, June 2008, pp. 361–366.

[3] J. B. Liu, A. Veneris, and H. Takahashi, "Incremental diagnosis of multiple open-interconnects," in *Proceedings. International Test Conference*, Oct 2002, pp. 1085–1092.

[4] V. D. Agrawal, D. H. Baik, Y. C. Kim, and K. K. Saluja, "Exclusive test and its applications to fault diagnosis," in *16th International Conference on VLSI Design, 2003. Proceedings.*, Jan 2003, pp. 143–148.

[5] B. Becker, M. Sauer, C. Scholl, and R. Wimmer, "Modeling unknown values in test and verification," *Formal Modeling and Verification of Cyber-Physical Systems*, pp. 122–150, 2015.