3-23-2018

Methods of Reverse Engineering a Bitstream for Field Programmable Gate Array Protection

Daniel J. Celebucki

METHODS OF REVERSE ENGINEERING A BITSTREAM FOR FIELD PROGRAMMABLE GATE ARRAY PROTECTION

THESIS

Daniel J. Celebucki, 2d Lt, USAF

AFIT-ENG-MS-18-M-018

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-18-M-018

METHODS OF REVERSE ENGINEERING A BITSTREAM FOR FIELD PROGRAMMABLE GATE ARRAY PROTECTION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Daniel J. Celebucki, B.S.E.E.

2d Lt, USAF

March 2018

AFIT-ENG-MS-18-M-018

METHODS OF REVERSE ENGINEERING A BITSTREAM FOR FIELD PROGRAMMABLE GATE ARRAY PROTECTION

THESIS

Daniel J. Celebucki, B.S.E.E.
2d Lt, USAF

Committee Membership:

Scott. R. Graham, PhD
Chair

Maj J. Addison. Betances, PhD
Member

Sanjeev. Gunawardena, PhD
Member

AFIT-ENG-MS-18-M-018

Abstract

Field Programmable Gate Arrays (FPGAs) are found in numerous industries including consumer electronics, automotive, military and aerospace, and critical infrastructure. The ability to be reprogrammed as well as large computational power and relatively low price make them a good fit for low-volume applications that cannot justify the Non-Recurring Engineering (NRE) costs associated with producing Application-Specific Integrated Circuits (ASICs). FPGAs however, have seen a variety of security issues stemming from the fact that their configuration files are not inherently protected.

This research assesses the feasibility of reverse engineering the bitstream format for a previously unexplored FPGA, as well as the utilization of the knowledge gained during that process to create a bitstream parser and perform a bitstream modification attack. The reverse engineering process utilizes Tool Command Language (TCL) scripts to automate the modification of various configuration options and then synthesize the resulting bitstream. Various configuration options for Input/Output Blocks (IOBs) are mapped to their respective locations in the bitstream and the encoding format for the configuration of several Look-Up Tables (LUTs) is discovered.

This information is then utilized to create a bitstream parser that takes a bitstream as an input and outputs configuration information for IOBs. Additionally, a bitstream modification attack is performed that changes the original design logic by modifiying the bitstream directly to change the configuration values of a LUT. Both the parser and bitstream modification attack are shown to work validating the information gained through the reverse engineering process.

Acknowledgements

I would like to thank my advisor, Dr. Scott Graham, for his advice and mentorship during the thesis process. I would also like to thank my committee and professors for helping me learn the skills needed to complete this reasearch. Finally, I would like to thank my family for their continued love and support which has allowed me to pursue my goals.

Daniel J. Celebucki

Table of Contents

Page

List of Figures

Figure Page

List of Tables

Acronyms

ASIC Application-Specific Integrated Circuit

CLB Configurable Logic Block

CMOS Complementary Metal-Oxide-Semiconductor

FPGA Field Programmable Gate Array

GUI Graphical User Interface

HDL Hardware Description Language

ICS Industrial Control System

IOB Input/Output Block

IP Intellectual Property

LED Light Emitting Diode

LPF Lattice Preference File

LUT Look-Up Table

mA milliamp

NCD Native Circuit Description

NRE Non-Recurring Engineering

RPM Revolutions Per Minute

SRAM Static Random Access Memory

TCL Tool Command Language

METHODS OF REVERSE ENGINEERING A BITSTREAM FOR FIELD PROGRAMMABLE GATE ARRAY PROTECTION

I. Introduction

1.1 Background and Motivation

FPGAs are used in the military, automotive, and consumer industries and their prevalence is increasing [7]. Although these devices continue to increase in capacity and speed [24], they still face a number of security issues [8, 3, 22, 15].

Reverse engineering is one of these issues. When considering FPGAs, reverse engineering usually consists of "transforming an encoded bitstream into a functionally equivalent description of the original design" [7]. However, the reverse engineering effort can also be partial, meaning the full functionality of the design is not reproduced but data from the bitstream such as encryption keys or LUT content is still extracted. Reverse engineering itself can be considered a security issue but it can also enable other security threats by making use of information gained during the reverse engineering process.

One such threat that can be enabled is bitstream modification or injection. This attack focuses on directly modifying the encoded bitstream in order to produce a change in the original design. Because the original design is not known, modifications to the bitstream are difficult, and usually result in a broken design or the design not loading. Bitstream injection was considered improbable before 2013 when Chakraborty demonstrated that presynthesized ring oscillators could be injected into the bitstream in locations that had not been utilized by the original design [3]. The

ring oscillators were used to dissipate power increasing the circuit operating temperature. While impressive, injecting into unused portions of an FPGA cannot be used to interact with the original design. This research focuses on utilizing the knowledge gained through reverse engineering to protect FPGAs against security threats as well as execute a more advanced bitstream modification attack.

1.2 Problem Statement

This research presents the process and results of reverse engineering a bitstream in order to show that this knowledge can be used to both protect or attack FPGAs. As FPGAs increase in prevalence, identifying security threats and ways to defend against these threats becomes increasingly important.

1.3 Research Objectives

The goal of this research is to assess whether reverse engineering the bitstream format for a LatticeECP3 LFE3-35EA-8FN484C FPGA can be used to protect or attack an FPGA. There are two main hypotheses. The first is that a parser can be created that can detect malicious modifications to a bitstream by utilizing the information gained by reverse engineering the bitstream. The second is that a bitstream modification attack that directly influences the original design of that bitstream can be carried out using information from reverse engineering that bitstream.

1.4 Organization

Chapter II provides background relevant to the understanding of this research. FPGA structure and programmability is discussed along with various security features and threats. Chapter III discusses the target system, assumptions, design decisions, and methodology to reverse engineer a bitstream, create a bitstream parser, and

perform a bitstream modification attack. Chapter IV discusses the results of the reverse engineering process, effectiveness of the bitstream parser, and the results of the bitstream modification attack. Chapter V concludes with a summary and a discussion of future objectives for continuing research.

II. Background

This chapter provides background information and context related to Field Programmable Gate Arrays (FPGAs). Information describing the composition of FPGAs as well as the process behind their configuration is provided.

2.1 Reverse Engineering

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction [4]. Reverse engineering is typically used by adversaries trying to gain knowledge about a system when they lack design information or details. Reverse engineering can be used on both hardware and software systems but the objective is typically different. When considering a hardware system, the objective is usually to duplicate the system. When reverse engineering software, the objective is usually to gain a design-level understanding of the system to aid maintenance, strengthen enhancement, or support replacement [4].

When considering FPGAs the objective of reverse engineering is usually to understand the bitstream format. This is because if the bitstream format is known, the netlist can be recovered [21, 2, 6]. Although the netlist is not the same as the Hardware Description Language (HDL) design file, the netlist contains all the components that make up the hardware design being implemented on the FPGA. If one can reverse engineer the bitstream format they can then use that knowledge to recover the netlist, often considered to be proprietary, and even referred to as Intellectual Property (IP).

2.2 Field Programmable Gate Arrays

FPGAs were first introduced in 1984 by Xilinx and over the next three decades increased in capacity and speed by factors of 10000 and 100 respectively [23]. Unlike traditional Application-Specific Integrated Circuits (ASICs), which are customized for a particular use, FPGAs are reprogrammable. This reprogrammability is accomplished using a combination of Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs) and interconnects. Figure 1 shows an example FPGA architecture highlighting the CLBs, IOBs, and interconnects. CLBs are made up of various digital circuits such as Look-Up Tables (LUTs), multiplexers, adders, and flip flops. The CLBs can then be configured to perform different combinational functions. The interconnects are used to connect the various CLBs to create the desired circuit. IOBs provide connections to external stimulus. The trade off for this reconfigurability is that ASICs are more compact and power efficient than an FPGA implementing the same circuit.

FPGAs are configured using a HDL which is synthesized into what is generally referred to as a 'bitstream'. The HDL code describes the structure and behavior of the circuit while the bitstream is a series of `0's` and `1's` which specify the configuration options of the CLBs, IOBs, and interconnects to produce that circuit. Each FPGA vendor has their own proprietary bitstream format and the details of that format are not usually released to the public.

Depending on the type of FPGA (Static Random Access Memory (SRAM), Flash, or Antifuse) the bitstream is programmed and stored in different ways [19]. SRAM FPGAs use internal volatile static latch cells to store configuration data and need to be programmed after every power cycle [24]. After they are powered on the bitstream is transferred from an external non-volatile memory source to the FPGA. Flash FPGAs use internal non-volatile memory to store the configuration data which means

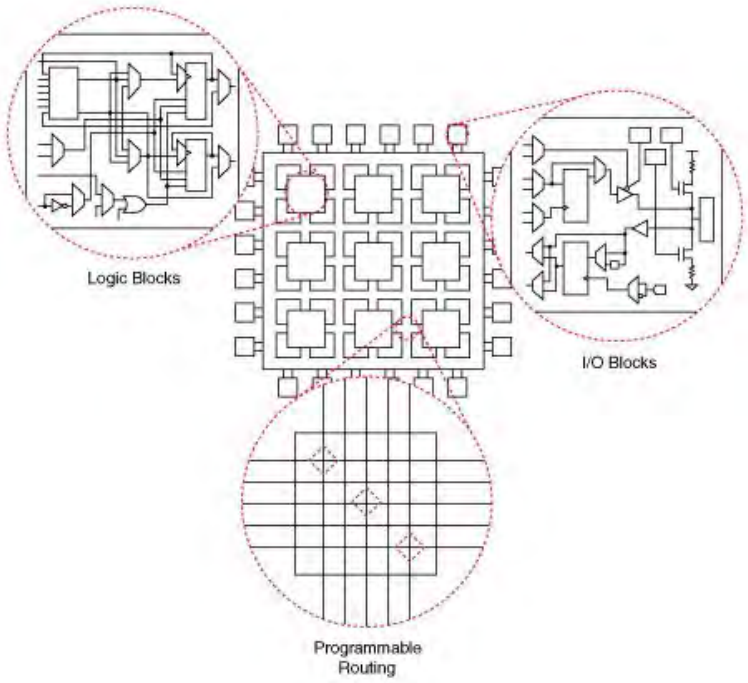


Figure 1. FPGA Architecture showing CLBs, IOBs, and Interconnects [20].

no external bitstream transfer is necessary. It also allows for faster power-on and information transfer through power-cycles [24]. Antifuse FPGAs program through a pulse which form low-resistance connections between the internal nodes making it one-time programmable. Regardless of the type of FPGA the configuration is largely the same. The bitstream's ones and zeros are used to tie high and low voltages to LUT initialization values, multiplexer selectors, switching matrix transistors, etc. Every configurable option on the FPGA is represented somewhere in the bitstream and is tied to high and low voltage states based on the values in different locations in the bitstream.

Configurable Logic Blocks.

CLBs are what allow the FPGA to implement custom logic. Although there can be some differences between the hardware implementation of a CLB depending on the

manufacturer of an FPGA, they commonly include LUTs, multiplexers, and flip-flops. Figure 2 shows an example CLB with two LUTs and two flip-flops.

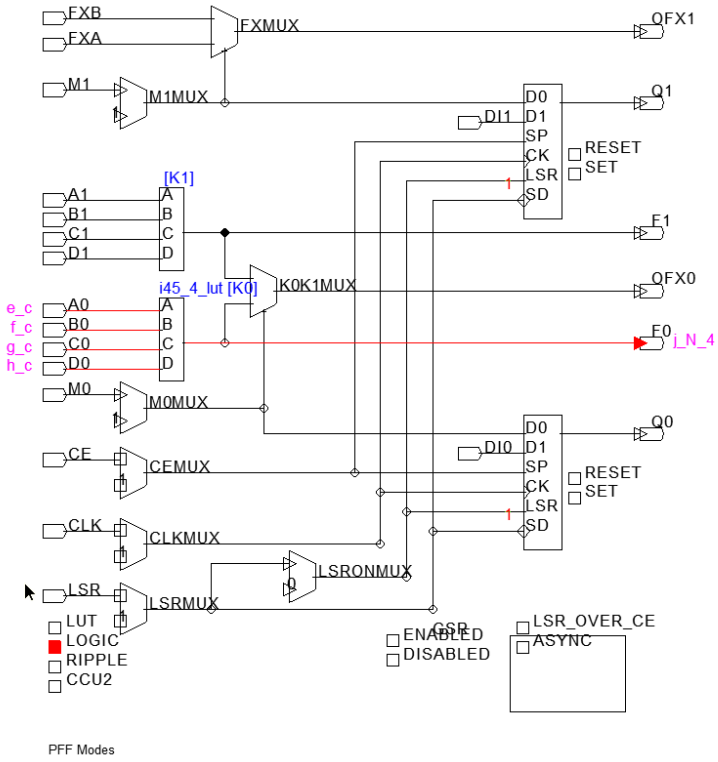


Figure 2. Example configurable logic block.

Instead of a traditional ASIC that uses hardware logic gates to implement the digital logic for the desired circuit, FPGAs utilize LUTs to implement the logic. Figure 3 shows an example of how a 2-input LUT utilizes multiplexers to implement digital logic. Instead of using an AND gate with a and b as the inputs, the LUT utilizes the inputs a and b as the selectors for the multiplexers and then, via the appropriate bits in the bitstream, initializes the inputs to the multiplexers as the output values for the truth table desired. This produces a circuit that is logically equivalent to an AND gate. If the user wants to implement a different circuit, the input values to the multiplexers can be changed allowing a new digital circuit without actually changing the FPGA hardware. LUTs trade space for reprogrammability in that the actual

hardware needed to implement the LUT is much larger than the hardware needed to implement the same digital circuit that the LUT is replicating. However, LUTs can be reprogrammed to implement any desired logic function as long as it fits within the bounds of the truth table. Designs that require more inputs than the LUT has available can be implemented by daisy chaining multiple LUTs together.

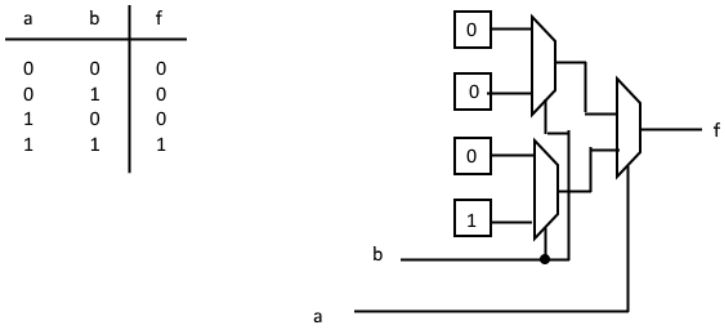


Figure 3. 2-input look-up table example.

Input Output Blocks.

IOBs are used to connect the internal logic of the FPGA to external components. At their most basic they consist of a physical pad that serves as the bridge between the FPGA and the rest of the system along with a number of multiplexers. Since the IOBs usually allow both input and output to the same physical pad, the choice of whether a certain pin will be an input or output is decided at configuration time. There are also other configuration options that determine the physical characteristics of the signal at the pin such as pullmode, slew rate, drive, etc. These configuration options vary from FPGA to FPGA. Figure 4 shows an example IOB that is configured as an output.

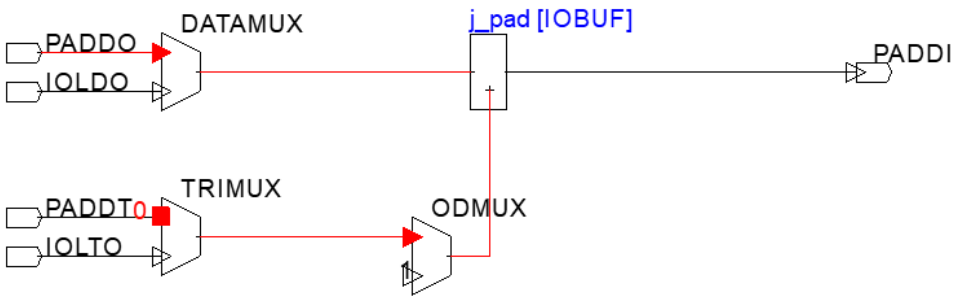


Figure 4. Example input-output block.

Switching Matrix.

The switching matrix is the largest portion of the FPGA (in terms of silicon area consumed) and is responsible for connecting the CLBs and IOBs to produce the desired digital logic circuit [9]. The reason that the switching matrix is so large is that nearly every CLB and IOB must be able to connect to each other. The large number of routes that need to be accommodated results in a large switching matrix. Figure 5 shows the routing paths of a simple digital circuit. The configuration of the switching matrix, which is determined by specific bits in the bitstream instantiates these routes.

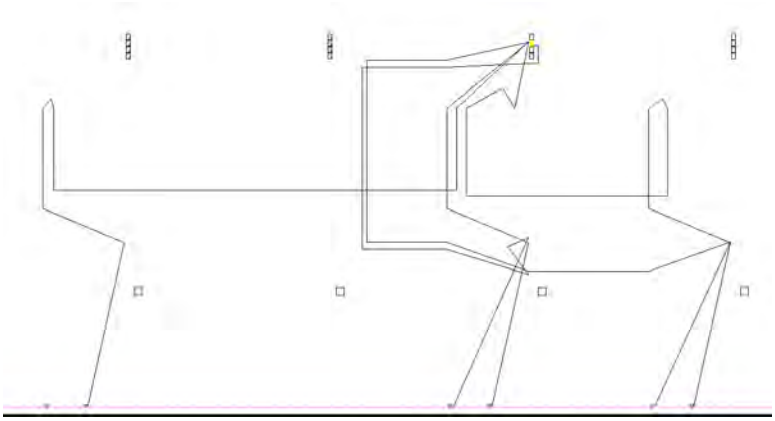


Figure 5. Virtual representation of routing paths within an FPGA

2.3 Bitstream Synthesis

An HDL file is transformed into a bitstream through a number of steps. Figure 6 shows the process that is used for most FPGAs. First the HDL code is *synthesized* into a netlist. The netlist contains the list of components in the circuit and the nodes they are connected to. The *map* function maps the components in the netlist to the components that are found on the FPGA the designer is compiling to. *Place* then takes the mapped list of components and selects the locations of those components on the FPGA. Since the FPGA will most likely have numerous copies of the same components, *place* decides which of those components will actually be part of the circuit. *Route* then makes the connections between all of the placed components on the board. After the circuit has been placed and routed it can be converted into a bitstream file that will configure the correct components and connections on the board to create the circuit.

Figure 6. The process for bitstream generation from HDL [14].

2.4 Field Programmable Gate Array Security

As FPGAs gained popularity, the issue of security rose in importance for both vendors and customers. Vendors wanted to protect their hardware designs and technology from being reverse engineered, copied, or modified by other vendors. They were also concerned with the protection of "soft cores" or HDL modules that were created to program an FPGA. The protection of soft cores is the primary security concern for customers because it protects their designs from being reverse engineered

or copied by competitors [8].

Unlike ASICs which are difficult to modify once they have been created and require sophisticated and expensive technology in order to reverse engineer the design, FPGAs merely need a configuration file which specifies the design to be implemented. This allows a competitor to start the reverse engineering process from a configuration file instead of the hardware. This is why bitstream formats are kept secret. If the format was publicly accessible, as soon as the competitor extracted the bitstream they could reverse the design. However, this does not protect the bitstream from reverse engineering, it only prolongs the process. Once a competitor figures out the format they can reverse the bitstream into a more useful file type.

FPGA vendors and customers have used a number of methods to protect their designs from competitors including obfuscation, encryption, and reconfiguration. For the rest of this thesis FPGA security will refer to the protection of soft cores from reverse engineering, copying, or modification rather than the protection of the actual FPGA hardware design unless specified otherwise.

Obfuscation.

One method that can be used to increase security is to obfuscate the design. Obfuscation is the process of intentionally modifying the description or structure of a circuit in order to conceal its functionality to make it more difficult to reverse engineer [10]. Although this method may not stop a competitor from reverse engineering the design, obfuscation may increase the cost of reverse engineering to the point where it is cheaper for the competitor to invest in creating their own design instead.

There are two approaches to hardware obfuscation, passive obfuscation and active obfuscation. Passive obfuscation alters the comprehensibility of the HDL code so that it is difficult for a human to understand but will still compile with the same

functionality. Active obfuscation on the other hand, alters the functionality of the circuit. Many times active obfuscation is key based where a key or sequence of keys must be applied to the input to unlock the normal function of the circuit. If the key is not applied the circuit will function incorrectly [10].

Encryption.

In order to protect the intellectual property of their customers, FPGA vendors offer a bitstream encryption feature on high-end models. Figure 7 shows an example of how bitstream encryption works. In this example a FPGA based network router is being updated. Both the designer and the FPGA share the same secret key. The designer encrypts the new bitstream and sends it to the router. The FPGA has an internal decryption engine and decrypts the bitstream after it is loaded from configuration memory. After decryption, it configures itself according to the new bitstream. Even if the bitstream is intercepted over the Internet or when the bitstream is being loaded from configuration memory, the attacker will be unable to decrypt the bitstream without the secret key [17].

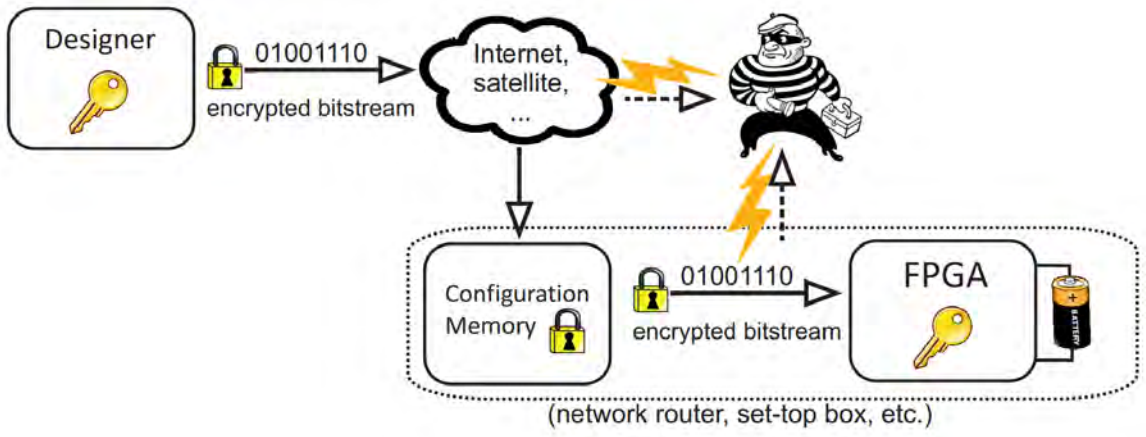


Figure 7. FPGA receiving encrypted bitstream from designer [17].

However, encryption alone is not sufficient for system security in a number of

situations including [11]:

1. Applications requiring FPGAs with low-energy and cost requirements or small form-factors. These FPGAs may not have encryption blocks because of their constraints.

2. Remote upgrades of the design using a new encryption key. In these cases the encryption keys must be sent with the bitstream leaving them vulnerable to an attacker who can intercept them.

3. Devices that remain in the field for many years. They are susceptible to physical attacks such as side channel analysis [17].

2.5 Field Programmable Gate Array Attacks

As vendors and designers have come up with different ways to protect their intellectual property, adversaries and competitors have come up with their own ways to reverse engineer, copy, and modify those designs. Security follows a cyclical nature with new attacks emerging followed by security measures to defeat those attacks, finally followed by new attacks and the cycle continues [8]. Two types of attacks that have been particularly noteworthy in the past decade are side-channel analysis attacks and implanting hardware trojans.

Side-Channel Analysis.

Side-channel analysis attacks exploit physical information leakage of operations executed within a device in order to extract secret data. Many times the secret data is related to the cryptographic key stored on the device used to decrypt the bitstream [17]. Side-channel analysis can come in a variety of forms depending on which external

characteristics are measured. Three types of side-channel analysis related to FPGAs are [8]:

1. **Power analysis:** Power is consumed in two ways in integrated circuits. Dynamic power consumption is due to Complementary Metal-Oxide-Semiconductor (CMOS) gates changing state according to the logic transition. Static power consumption is due to current flowing between the source and drain terminals and through gate oxide. This is referred to as "gate leakage". By analyzing the electrical current patterns of an integrated circuit, information about the data it is processing may be revealed to the attacker.

2. **Electromagnetic analysis:** Electromagnetic fields are caused by current changes during execution of a function. These fields can be detected outside of the device using finely tuned antennas and analyzed to uncover the secret data.

3. **Timing analysis:** The timing of different functions such as conditional branching, memory access, and algorithmic operations are often related to the key state during cryptographic operations. An example of this would be comparing a password one character at a time. If the function took different amounts of time for a match and miss then the attacker could determine the password.

Hardware Trojans.

Hardware Trojans are malicious, hard-to-detect hardware modifications. They have become a potent threat to ensuring trustworthiness of integrated circuits due to outsourcing steps in the manufacturing process [3]. They can be used for a variety of functions including decreasing the lifetime or reliability of the device, functional failure, or leakage of secret keys. They are also a significant threat to FPGAs since the bitstream used to configure the device could potentially implement a Trojan.

Without the source code used to generate the bitstream it can be very difficult to detect their presence. Additionally, it is possible to insert a Trojan into an FPGA design by modifying the bitstream [3]. In this case, the attacker does not need any knowledge of the design to insert the Trojan. Instead the Trojan is inserted at a location in the bitstream that was not being utilized.

Bitstream Modification.

Bitstream modification can be performed by both an adversary that intercepts the bitstream, or an insider. Consider the situation where an Industrial Control System (ICS) company is licensing the IP that it uses for its FPGAs. If an adversary can intercept the bitstream, it could be modified and the ICS company would be unable to tell since they only receive a bitstream file. Chakraborty showed that this is a real threat since a bitstream can be modified to introduce hardware trojans without actually knowing the HDL used to create the bitstream [3]. In this case they inserted ring oscillators to increase the temperature of the FPGA thereby accelerating its aging. This attack could be implemented by an insider that is responsible for loading the bitstream onto the board, or by an adversary that intercepts the original bitstream and delivers the modified bitstream to the company.

An adversary that is able to reverse engineer the target FPGA could go farther by gleaning information about the bitstream and then making changes to the bitstream directly. This information allows the adversary to make more effective changes without having to understand the totality of the design. If the adversary is able to completely reverse engineer the bitstream file then they will have absolute control in understanding and changing the bitstream.

Bitstream Reverse Engineering.

Bitstream reverse engineering is a significant threat to the security of the intellectual property of designers. Since the bitstream is a string of `0's` and `1's` that configure the components on the board, until the competitor can determine how the bitstream is configuring those components the design is still protected. However, researchers have shown that given enough time, a bitstream can be reverse engineered into the netlist which describes the design [21, 6].

Once the netlist has been generated the design can be understood, modified, and synthesized to run on other FPGAs. However, the process is time intensive. In order to isolate a configuration option many bitstreams must be generated, with each containing slight differences. The bitstreams can then be compared to see how configuration changes affect the makeup of the bitstream. This process then has to be repeated over and over again for each configuration option. Additionally, some configuration options are difficult to isolate because a change to a certain configuration option can also affect other configuration options.

Covert Channels.

A covert channel allows two cooperating entities to communicate secretly, in violation of a security policy, by manipulating shared resources [16, 12]. This is similar to a side-channel however, a side-channel leaks information to other parties, and does not require the cooperation of malicious entities. Covert channels are further divided into storage channels and timing channels. Storage channels communicate by modifying a storage location such as a hard drive or system memory. Timing channels communicate by performing operations that affect the real response time observed by the receiver. To show a covert channel example consider the case where a vehicle with an FPGA wants to fool an emissions test. The emissions test will most likely

use the same sequence of inputs to test the vehicle such as throttle the engine to 2000 Revolutions Per Minute (RPM) and hold for 3 minutes, increase to 4000 RPM for one minute, etc. In order to fool the test a storage channel between the IOBs and a set of CLBs could be implemented. The IOBs could write the values of the RPM values to some location in memory that is then checked by the CLBs controlling ignition timing and other factors that affect emissions. If the CLBs see a certain sequence of RPM values they would know that they are in the middle of an emissions test and possibly change ignition timings or other engine factors in order to reduce emissions. A timing channel could be implemented to achieve the same feat by increasing or decreasing the delay for the signal between the IOBs and the CLBs to communicate different RPM values. If a certain sequence of delays is recognized the outputs are then changed to produce fewer emissions.

III. Experimentation Methodology

This chapter explains the methodology used to reverse engineering an Field Programmable Gate Array (FPGA) bitstream format as well as utilize the knowledge gained through the reverse engineering process to perform a bitstream modification attack.

3.1 Assumptions

There are a number of assumptions that must be made in order for this research to succeed. The first is that the bitstream is either not encrypted, or that the decrypted bitstream can be recovered somehow. Encrypted bitstreams impede the ability to parse information about the bitstream or execute a bitstream modification attack because the mapping of the configuration options to the bitstream is no longer constant. Reverse engineering the bitstream format using encrypted bitstreams would be much more difficult and even if some progress was made, changing the encryption key would negate any work done.

The second assumption that is made is that the bitstream is not obfuscated in any way. This means that the bitstream file is exactly the file that is loaded into the configuration memory in order to program the FPGA. If this bitstream was obfuscated and went through some sort of decoder hardware before being loaded into the configuration Static Random Access Memory (SRAM) the reverse engineering process could be significantly harder.

These are both valid assumptions to make because not all FPGAs offer encryption options and even those that do have had their encryption broken [17, 18]. Additionally, assuming that the bitstream is not obfuscated is completely valid for most situations because the common situation is to have the bitstream directly loaded into the

configuration memory instead of passing it through some decoding hardware [25, 1].

3.2 Target System

The target system for this paper was the LatticeECP3 LFE3-35EA-8FN484C FPGA. Although this work could have been performed on any FPGA, this one was chosen for a number of reasons. Both Xilinx and Altera have had numerous scientific research articles published on their FPGAs with both having been explored to some degree in regards to reverse engineering their bitstream [21, 22]. Lattice, on the other hand has seen much less attention on anything other than the extremely small iCE40 FPGAs [5]. This allows for comparisons to be made between the reverse engineering process for the Lattice FPGAs and the processes for the Xilinx and Altera FPGAs. Additionally, since any FPGA could be studied and we had decided on a Lattice FPGA due to the reasoning above, we decided to pursue a investigate a mid-grade FPGA with enough computational power to implement many different designs. To reverse engineer a bitstream, the primary tool needed is the synthesis software that transforms a Hardware Description Language (HDL) design into a bitstream. Having the physical device, with the actual FPGA is only needed to test bitstream modification attacks.

All bitstreams were synthesized using the Lattice Diamond software, version 3.9.1. with the specified FPGA as the target device, and utilized the Lattice Synthesis Engine for synthesis of the bitstreams. The synthesis of bitstreams was performed using Tool Command Language (TCL) scripts that were evaluated by the `pnmain.exe` application. This allowed for the scripting of bitstream generation.

The Lattice Diamond software is free to download from Lattice Semiconductor and allows HDL code to be synthesized into bitstreams for various Lattice FPGAs. A free license allows synthesis for lower end FPGA models and purchasing a license

allows synthesis for higher end models. The tool organizes designs using projects and implementations and has a number of views to show information about the project. Figure 8 shows an example view with the HDL editor as the main view, the implementations within the project shown on the left side and the TCL Console shown below. Figure 9 shows the process view instead. Projects share the same target FPGA and can contain different implementations. Implementations are groups of files such as HDL, Lattice Preference File (LPF), analysis, and programming files that are used to synthesize a bitstream for the target device. A project can contain any number of implementations but only one implementation can be active at a time. Additionally, only one process can be running on a project at any given time.

Projects can be created using the New Project Wizard shown in Figure 10 or by using TCL commands. When actions are selected within the Graphical User Interface (GUI) they are then translated into TCL commands and then evaluated in the console. Figure 8 shows an example of this happening. After the new project selections were made using the GUI in Figure 10 the following line appears and is executed in the TCL console which can be seen in the bottom pane of Figure 8:

```
prj_project new -name "clbtest4" -impl "impl1" -dev LFE3-35EA-8FN484C
-synthesis "lse"
```

This is the TCL command that is generated by the GUI and is evaluated by the Lattice Diamond software to create a new project. Additionally, projects can be modified and synthesized without ever using the GUI by running TCL scripts through the command line TCL console. Projects can be created, source files can be added or removed, and bitstreams can be generated all without interacting with the GUI. This was pivotal to the reverse engineering effort because manually configuration for producing bitstreams would be very time consuming and likely suffer from numerous human input errors.

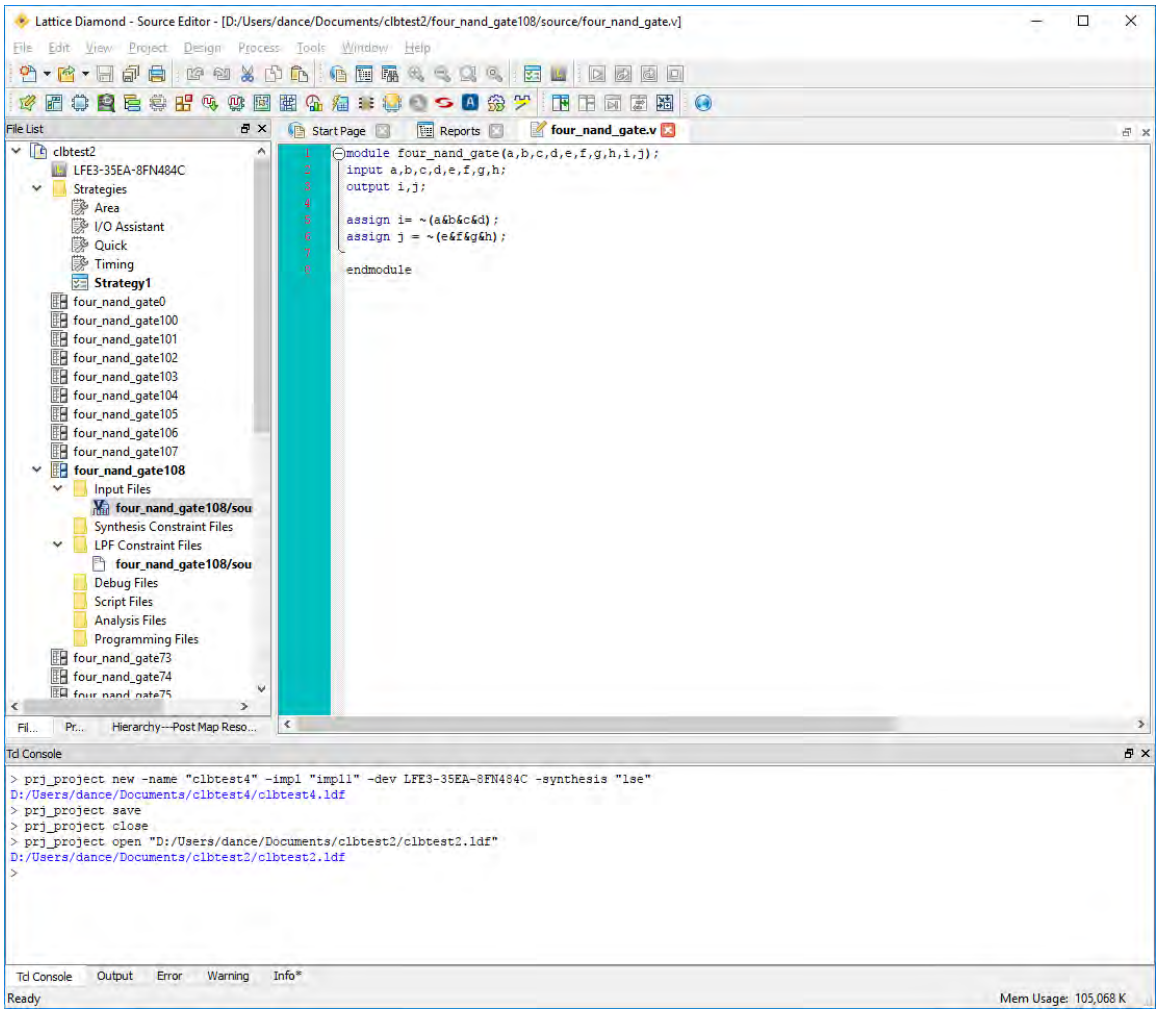


Figure 8. Lattice Diamond software GUI showing HDL editor, File List view, and Tcl Console.

3.3 Bitstream Reverse Engineering Process

This section serves to describe the general process used when reverse engineering certain configuration options for the Lattice LFE3-35EA-8FN484C FPGA. The first step was to downselect to a specific configuration option to reverse engineer. This could be a specific pin set as an input or output, a logic function initialized in a Configurable Logic Block (CLB), a certain intersection in the switching matrix, etc. Upon selecting the configuration option to be investigated, the option was exercised

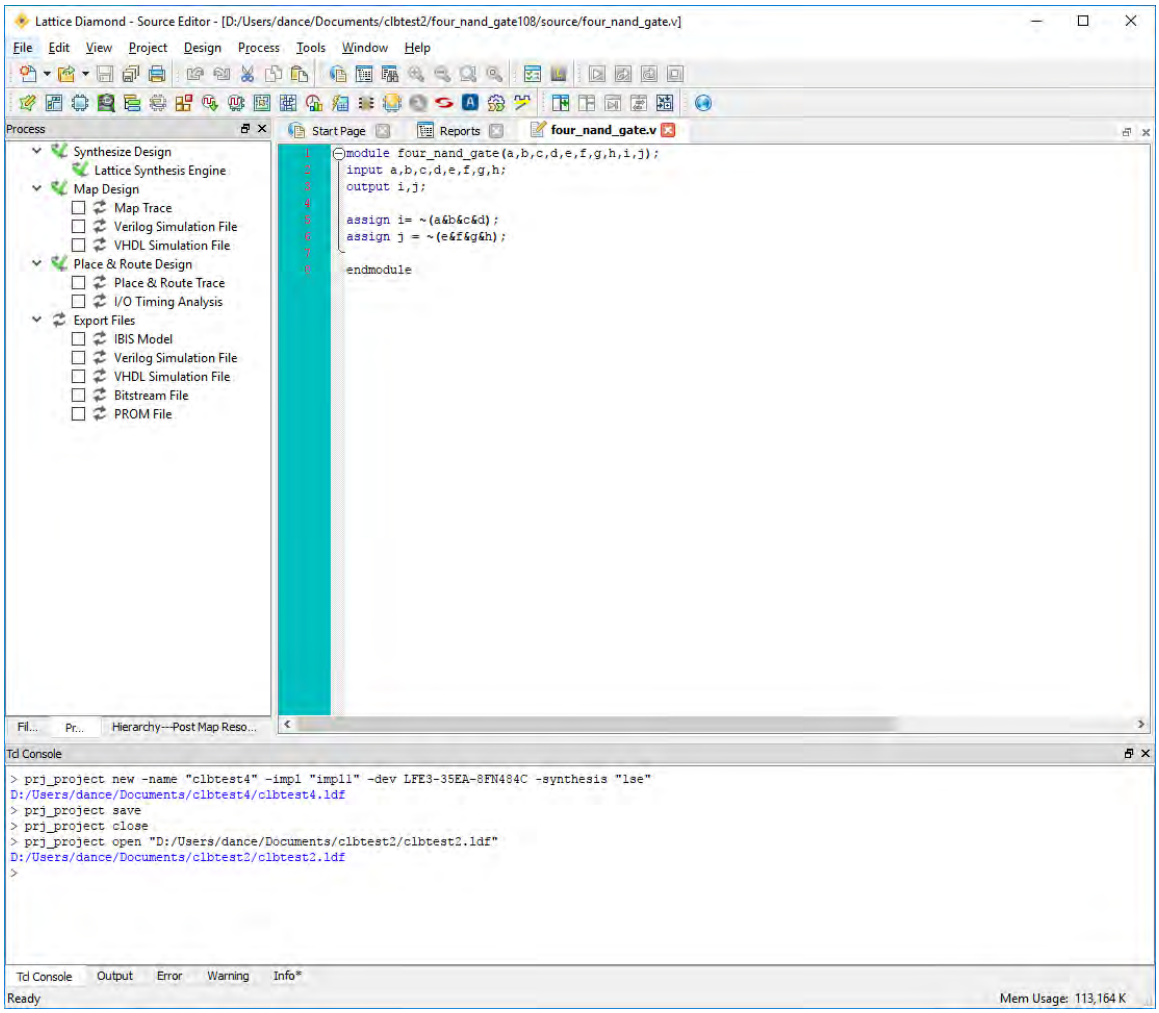


Figure 9. Lattice Diamond software showing the Process view.

through all possible values, generating a corresponding bitstream for each value. This was done using TCL scripts that would modify either the LPF or the HDL file and then synthesize the bitstream. These scripts all followed a similar pattern. The TCL script would open a Lattice Diamond project that already had a prepared HDL file and LPF, copy the HDL design and LPF into a new implementation and set it as the active implementation, modify the LPF or HDL file, synthesize the bitstream, and then repeat. Listing A.1 shows an example of one of these TCL scripts that was used to generate bitstreams for the pullmode option for each of the pins. A single script

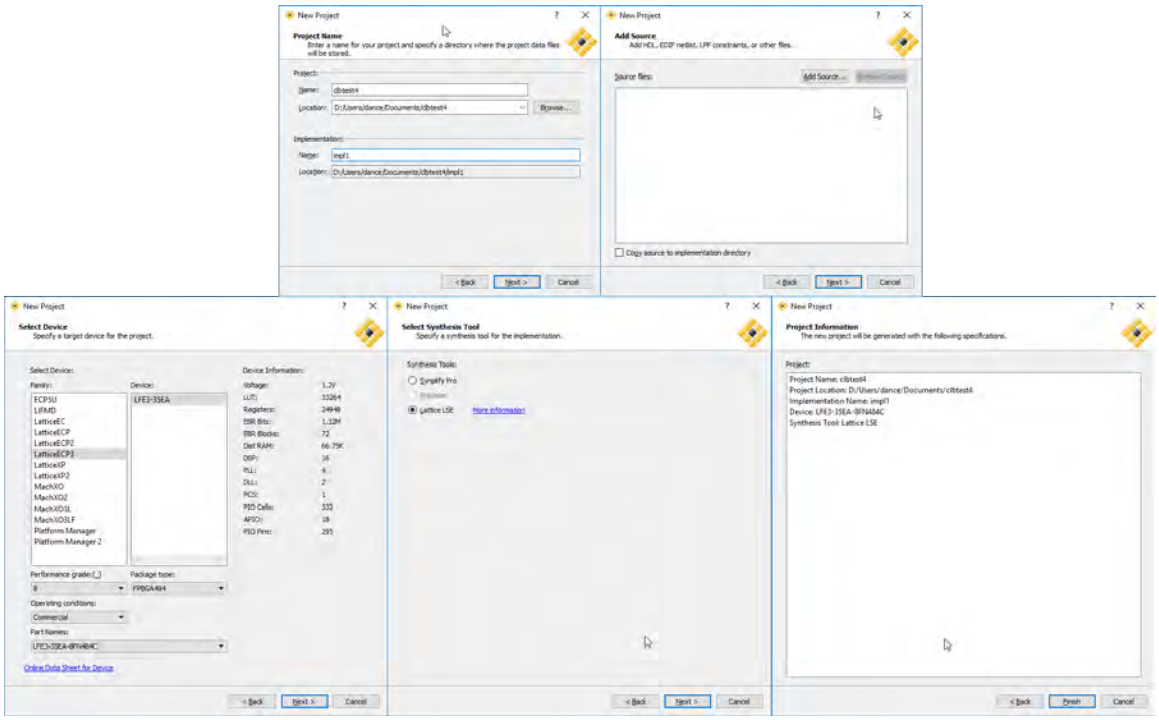


Figure 10. Lattice Diamond New Project GUI.

could exercise configuration options for any number of components on the board. The script would name and relocate the bitstream files as it synthesized each one so that all files were in a single location. Figures 11 and 12 show the script executing and the resulting output bitstream files. The TCL scripts were executed via batch files that would call the TCL file so that groups of TCL scripts could be executed sequentially.

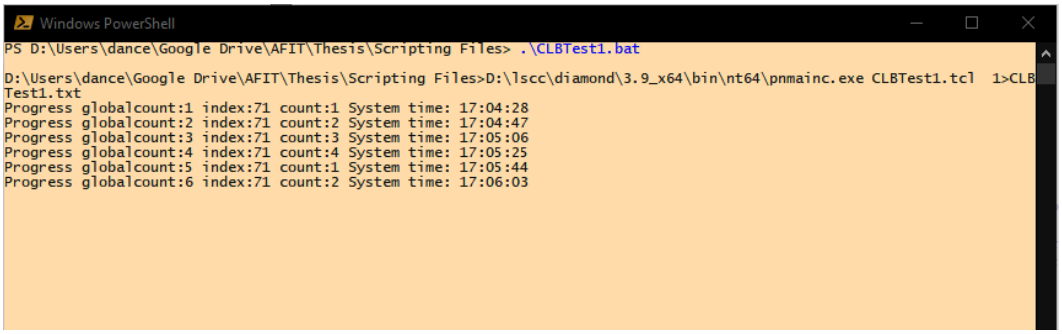


Figure 11. TCL script running generating bitstream files for analysis.

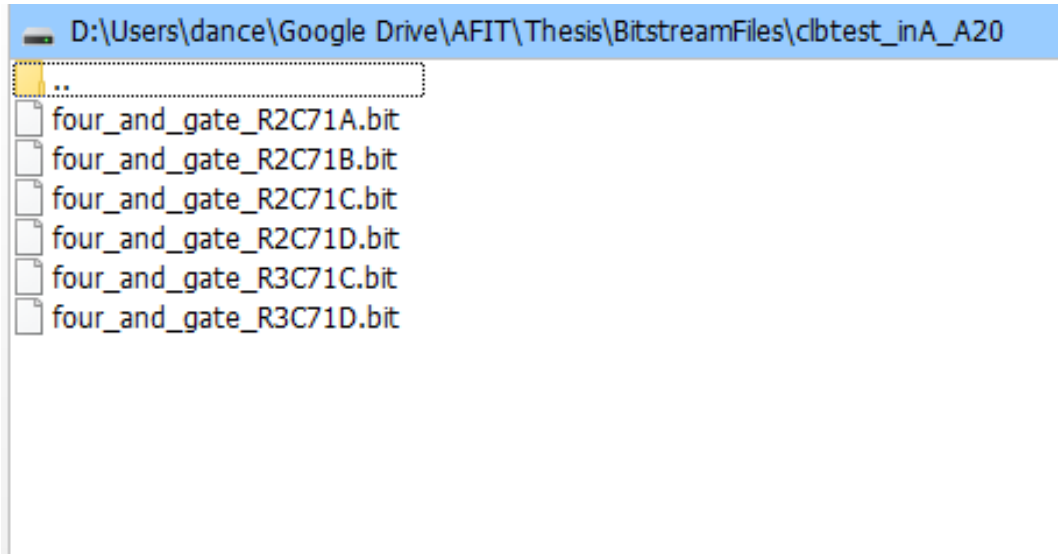


Figure 12. Bitstream files generated, renamed, and moved by the TCL script.

After the bitstreams were generated they were analyzed in an Ubuntu 14.04 virtual machine, harnessing the many advanced shell scripting commands. The bitstreams were analyzed using the program found in Listing A.2 which takes bitstream files as inputs and outputs the location of differences between these files and the values at those locations. Figure 13 shows an example of this program being used. The two `.bit` files are the bitstreams that are compared and the numbers in the leftmost column represent locations in the bitstreams where the two files differ. The hex values on the same row as one of those locations represents the values of the bitstreams in hex at those locations. After comparison, the binary of bitstreams was usually analyzed and compared to each other. This made it easier to see trends that were more difficult to notice when observing the hex outputs. Figure 14 shows an example of the bitstream printing tool found in Listing A.3 being used to print the binary from index 0 through index 20. The tool is able to print both the binary or hex depending the flag specified.

While analyzing these bitstreams it became obvious that the configuration options

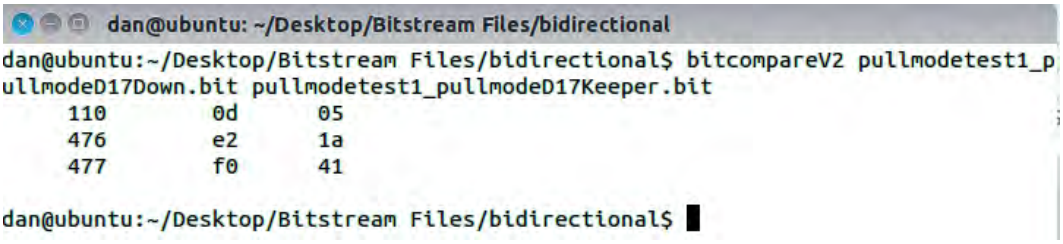


Figure 13. Results of bitstream comparison tool comparing two bitstreams.

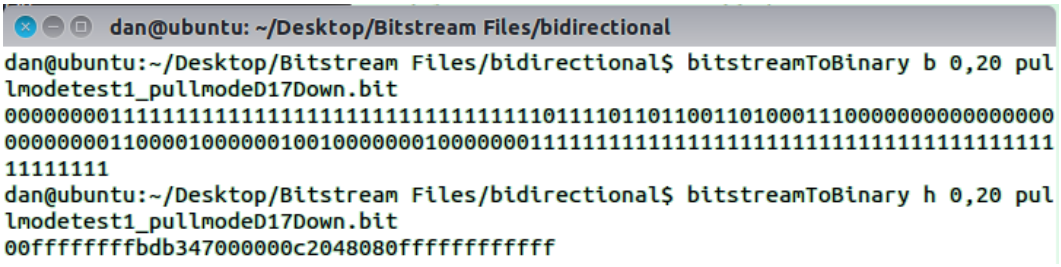


Figure 14. Results of bitstream printing tool outputting binary for a bitstream.

that could be exercised in isolation should be reversed first. If an option affected other parts of the design it was difficult to isolate which portions of the bitstream were responsible for that change because many indices could change at once. For example, switching a pin from an input to an output could result in greater than 50 changes in the bitstream because the switching matrix was also affected. Isolating which parts of the bitstream were responsible for the shift from input to output and which portions were the switching matrix was very complicated, involving many bitstreams, and lots of comparison and contraction.

However, much like corner pieces in a jigsaw puzzle, configuration options that manifested few changes in the bitstream when exercised could be used as figurative footholds to help reverse other configuration options by observing similar patterns or eliminating certain locations in the bitstream based on previously reversed options.

3.4 Input/Output Blocks

This section describes the process of mapping the relationship between the bitstream file and the configuration of the Input/Output Blocks (IOBs).

IOBs have two main advantages when considered in a reverse engineering perspective. The first is that there are far fewer of them than either CLBs or the switching matrix. There are 295 pins on the target FPGA while there are 33264 Look-Up Tables (LUTs) as shown in Figure 10. Additionally, the IOBs and CLBs make up a very small percentage of the board when compared to the switching matrix [9]. Their smaller number means that it takes less time to synthesize bitstreams for all of them in order to compare how the configuration options manifest differently in the bitstreams.

The second advantage is that their configuration options can be changed very easily using either the spreadsheet view shown in Figure 15 or the LPF. The spreadsheet view shows all of the inputs and outputs for a given design and their configuration options. These options can be changed by double-clicking an option and selecting one of the given values. These changes are then translated into commands that are written to the LPF.

Being able to change configuration options directly ensures that the changes that manifest in the bitstream are both isolated and reflective of the actual configuration option. Consider the case of the LUTs which will be explained in more detail in a future section. There are two ways that the configuration values of a given LUT could be changed. The first would be to write a variety of HDL files using different logic operators that are then translated into LUT configuration values. The second is to initialize those LUTs using supported HDL primitives and selecting the configuration values specifically. Method two is better suited for the reverse engineering effort because the bitstreams will definitely reflect the configuration values specified. Method

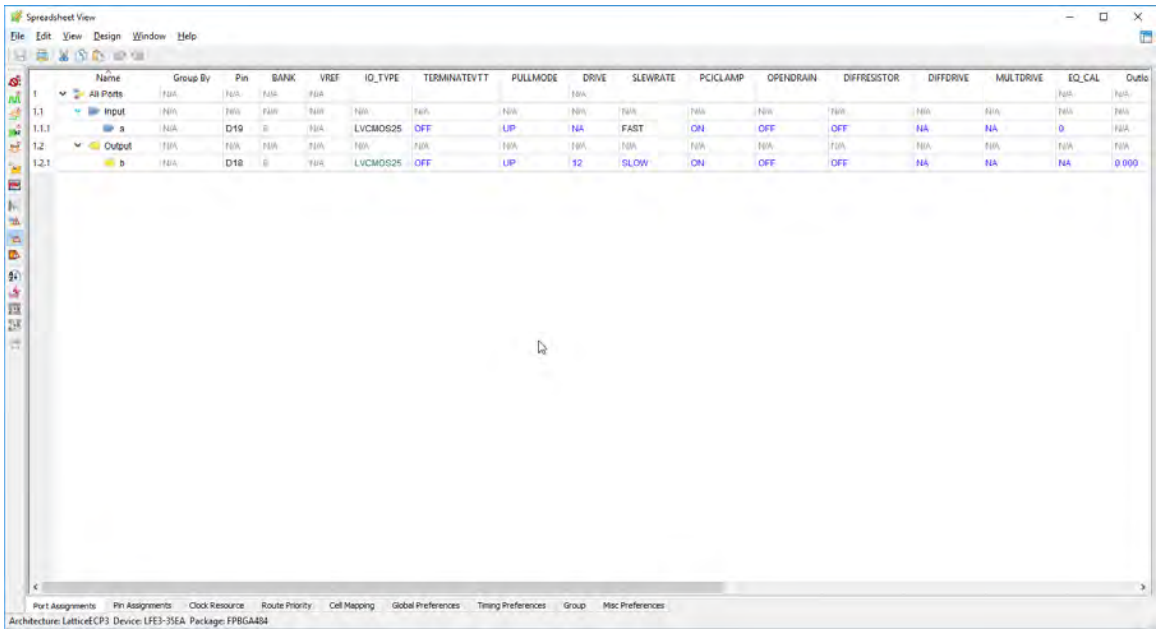


Figure 15. Lattice Diamond Spreadsheet View.

one on the other hand leaves the relationship obscured because the synthesis engine initializes the LUTs. If the synthesis engine is performing optimizations it is very difficult to know if the changes in the bitstream are actually reflective of the desired configuration option changes. Additionally, since changes made to the configuration options in the spreadsheet view were present in the LPF, modifying the LPF directly enabled automated bitstream generation.

There were three main goals when reverse engineering the IOBs. The first was to map all of the configuration options for each pin to their respective indices and values in the bitstream file. The second was to determine whether a pin was an input or output based on the bitstream file. And finally, to determine whether a pin was connected to logic blocks in the design. The following sections describe the methodology for reversing that configuration option.

Pullmode.

The pullmode is responsible for describing how the signal will be interpreted at the pin. The pullmode can be set to four options:

- **Up**: Input is attached to a pull-up resistor, i.e., the pin is tied to logical "1".
- **Down**: Input is attached to a pull-down resistor, i.e., the pin is tied to ground or logical "0".
- **Keeper**: Neither pull-up nor pull-down. Drives a weak 0 or 1 to match the level of the last logic state present on the pad to prevent the pad from floating.
- **None**: The input is not set to any of these modes.

Each of these options can be toggled using either the spreadsheet view or by modifying the LPF directly.

For the remainder of this document an index refers to the byte address in the bitstream after the header. When discussing changes in a bitstream due to a design change both the number of indices that change as well as the locations within those indices are discussed. A change in the pullmode of a pin had between 3 and 6 indices of change reflected in the bitstream. This was especially isolated when compared to other design changes such as moving the location of a CLB which could result in 70 - 100 indices changing. This isolation made pullmode a key candidate for reverse engineering.

Satisfied that this configuration option was sufficiently isolated, a TCL script synthesized bitstreams for every pullmode option for every pin, with each pin set as an input and then again with each pin set as an output. The pseudocode for that script is shown below. The objective was to determine which indices were responsible for the pullmode option for each pin as well as answer whether there was a common

pattern used for every pin to identify the pullmode. The hypothesis was that each pin would have a different location in the bitstream that would store its configuration options and that the values at those locations would each follow the same pattern in terms of representing the pullmode in the bitstream.

Algorithm 1 Pullmode Bitstream Generation

1: **for** Pin **p** in all I/O Pins **do**
2: **for** pullmode **val** in UP,DOWN,KEEPER,NONE **do**
3: Replace IOBUF line in LPF with "IOBUF PORT "a" PULLMODE=**val**"
4: **end for**
5: Replace Location line in LPF with "LOCATE COMP "<input/output pin name>" SITE **p**"
6: **end for**

After the script generated the 2296 bitstreams, they were then compared to find the indices in the bitstream responsible for the pullmode configuration option for each pin. As shown in Figure 16, relatively few indices changed for each pin. Each text file in the picture was generated by comparing the four bitstreams (pull-up, pull-down, bus keeper, and none) that were generated for each pin and listing all of the indices in the bitstreams that were different from any of the other bitstreams. In each text file, the first column with hex values refers to the values in the bitstream associated with pullmode pull-up, the second refers to pull-down, the third refers to bus keeper, and the fourth column refers to none. The numbers on the leftmost column are the indices in the bitstream where the changes occurred. So for instance, when comparing the four bitstreams that were generated for pin A2 when set as an input, the only differences between the four bitstreams appear in bytes 429, 476, and 477. It should be noted that indices 476 and 477 appear in almost every one of the listed comparison files. Recall that each index points to a specific byte in the bitstream. If any bit in

that byte is different, then the value at that index will change. Because we are searching for specific bits, it may be necessary to examine the values at the bit level rather than the byte level. However, with the bitstreams generated it was impossible to know whether all of the indices listed for each pin were necessary to configure the different pullmode options or if just a subset of the indices shown for each pin was necessary.

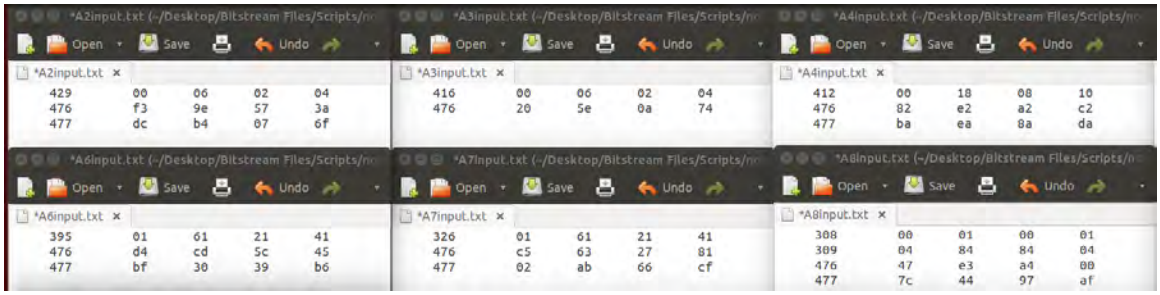


Figure 16. Indices in the bitstream responsible for the pullmode configuration option for various pins.

To solve this problem the bitstream generation script was again executed with different logic designs mapped to different portions of the FPGA. The reasoning for this was to isolate the indices responsible for the pullmode configuration option. Based on the hypothesis above, if the same generation script was executed with different combinational logic designs and mapped that logic to different portions of the FPGA, the indices that are responsible for pullmode configuration must have the same value across all of the different runs while indices that are affected by the switching matrix and CLBs would change. The following gates and placements were used.

1. 1 Input NOT Gate at R2C73D
2. 1 Input NOT Gate at R23C53A
3. 2 Input AND Gate at R3C70B
4. 1553 Encoder Placed by the compiler

When exploring which designs and placements were necessary to isolate the indices, the need to vary the CLBs in sufficiently diverse ways became clear. This was achieved by using a NOT gate, an AND gate, and the Intellectual Property (IP) core for a MIL-STD-1553 encoder. The encoder represented a larger design while the simple gates represented smaller designs. Each of the gates were then placed in different slices within CLBs on different portions of the FPGA and the 1553 encoder was placed by the tool. This proved to be enough variation initially and if none of the indices expressed different values additional variation could be introduced before considering that all of the indices listed were necessary to represent the pullmode configuration. After the bitstreams were generated for the additional logic designs, the bitstreams for each pin were compared and printed to text files. This comparison resulted in four comparison files for each pin created using the four bitstreams generated for each logic design. The four comparison files for each pin were then collated into a single text file in order to see how the indices and the values of those indices changed as the same pin exercised the same pullmodes across 4 different designs. Examples of these files can be seen in Figures 18 and 19. It should be noted that the indices never changed between different logic designs but the values at those indices sometimes changed. The indices whose values did not change between designs were assumed to be responsible for the pullmode configuration option and were examined more thoroughly.

To answer the question of why not just assume that all indices listed are responsible for pullmode, there are a few reasons that suggest otherwise. Firstly, the number of indices that changed for each pin when comparing bitstreams for each of the pullmodes was not constant. Some pins had only 3 indices change where as other pins had 6 indices change. Why would a designer decide to use more indices to represent the same change in different pins? Additionally, each pin had indices that were different

but some pins also had indices that were the same. Why would a designer have the same information located in two separate places when creating larger files will increase the time needed to configure the FPGA and also increase the complexity of the process used to parse the bitstream? It seemed more likely that some indices were being changed as a result of some other variation that was changing as a result of changing the pullmode.

Table 1. Pullmode Groups

Group Number	Indices
Group 1	476 or 477
Group 2	440404 or 440405
Group 3	667296 or 667297
Group 4	895054 or 895055
Group 5	Offset Pattern: 1, 2, 433, 434, 435
Group 6	Offset Pattern: 427, 428, 433, 860, 861

After collating the comparison files for each pin and noticing that the indices of change were the same, the indices changing for each pin were printed and then sorted numerically. The pins were then organized into 6 groups based on the indices responsible for their changes. Any pins that shared the same indices were grouped together as well as pins that shared a pattern in the offset of their indices. The grouping is shown in Table 1. Group 1 consisted of all pins with changes in indices 476 or 477. Group 2 consisted of all pins with changes in indices 440404 or 440405. Group 3 consisted of all pins with changes in indices 667296 or 667297. Group 4 consisted of all pins with changes in indices 895054 or 895055. Group 5 consisted of all pins not in one of the previous groups with an offset pattern of 1, 2, 433, 434, 435. Pin V5 is an example with its indices shown in Figure 17. Its second index of change 887693 is 1 greater than its first index of change 887692. Additionally, its third index of change is 2 greater than the first, its fourth index is 433 greater than the first index, etc. Finally, Group 6 consisted of all pins not in one of the first four

groups with an offset pattern of 427, 428, 433, 860, 861.

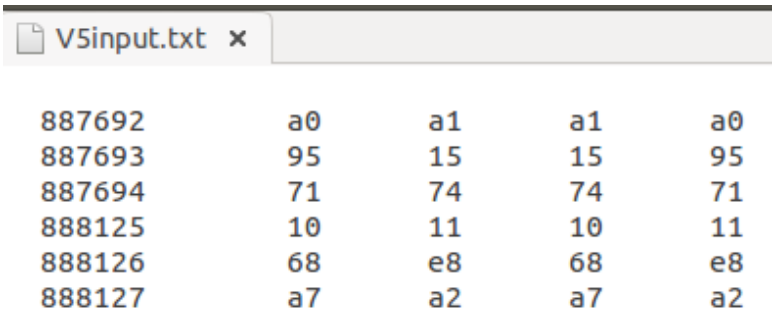


Figure 17. Indices related to changes of the pullmode for pin V5.

Group 5 and Group 6 may seem as if the criteria for the pins was arbitrarily picked. However, these pins did not lend themselves to an easy partitioning in the same way that Groups 1 through 4 did. These pins behaved differently than Groups 1 through 4 in that there were always 6 indices of change and the values at those indices did not change when different logic designs and placements were used. Figure 18 and Figure 19 show these differences. Pins A2, K1, R17, and AA4 are from Groups 1, 2, 3, and 4 respectively. Pins V5 and AB20 are from Groups 5 and 6 respectively. Pins in the first four groups always have 3 or 4 indices of change, share common indices with other pins in the group, and the values at those indices sometimes change depending on the design logic or placement. Pins in Groups 5 and 6 always had 6 indices of change and the values of those indices did not change with different design logic or placements. Since there wasn't an apparent strategy to partition the pins not in the first four groups, the offsets of the indices of each of the pins was calculated. Two clear patterns emerged. The remaining pins either had an offset of 1, 2, 433, 434, 435 or 427, 428, 433, 860, 861. These two offsets were then used to separate the pins into Groups 5 and 6. The pins were then analyzed as groups to find the relationship between the pullmode options and the representation in the bitstream.

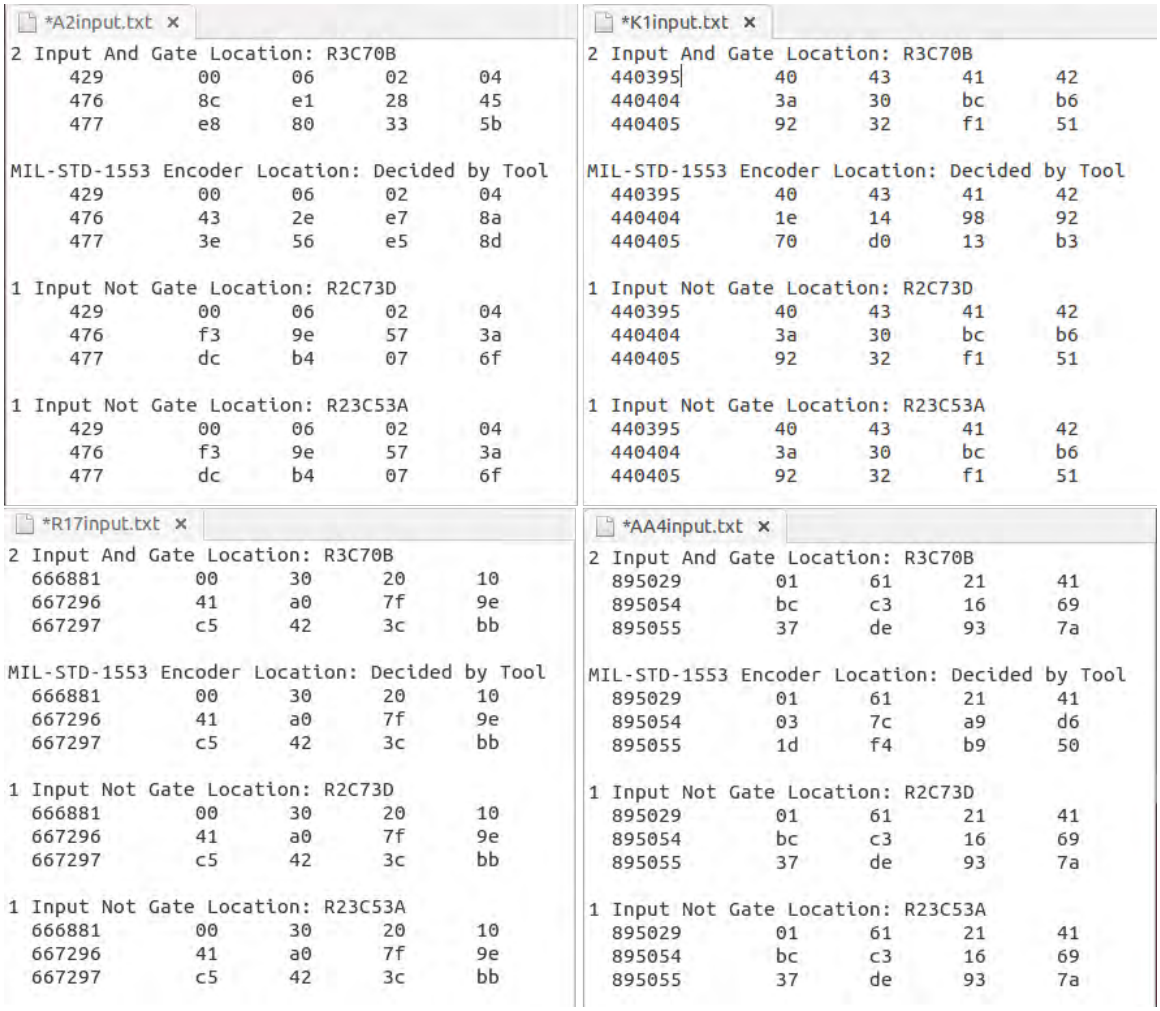


Figure 18. Pins Groups 1 through 4 showing the differences in changing indices.

*V5input.txt

```
2 Input And Gate Location: R3C70B
  887692        a0      a1      a1      a0
  887693        95      15      15      95
  887694        71      74      74      71
  888125        10      11      10      11
  888126        68      e8      68      e8
  888127        a7      a2      a7      a2

MIL-STD-1553 Encoder Location: Decided by Tool
  887692        a0      a1      a1      a0
  887693        3f      bf      bf      3f
  887694        d2      d7      d7      d2
  888125        10      11      10      11
  888126        22      a2      22      a2
  888127        0a      0f      0a      0f

1 Input Not Gate Location: R2C73D
  887692        a0      a1      a1      a0
  887693        95      15      15      95
  887694        71      74      74      71
  888125        10      11      10      11
  888126        68      e8      68      e8
  888127        a7      a2      a7      a2

1 Input Not Gate Location: R23C53A
  887692        a0      a1      a1      a0
  887693        95      15      15      95
  887694        71      74      74      71
  888125        10      11      10      11
  888126        68      e8      68      e8
  888127        a7      a2      a7      a2
```

*AB20input.txt

```
2 Input And Gate Location: R3C70B
  887266        f0      f8      f8      f0
  887693        95      5b      5b      95
  887694        71      14      14      71
  887699        f0      f8      f0      f8
  888126        68      a6      68      a6
  888127        a7      c2      a7      c2

MIL-STD-1553 Encoder Location: Decided by Tool
  887266        f0      f8      f8      f0
  887693        a6      68      68      a6
  887694        42      27      27      42
  887699        f0      f8      f0      f8
  888126        7a      b4      7a      b4
  888127        6f      0a      6f      0a

1 Input Not Gate Location: R2C73D
  887266        f0      f8      f8      f0
  887693        95      5b      5b      95
  887694        71      14      14      71
  887699        f0      f8      f0      f8
  888126        68      a6      68      a6
  888127        a7      c2      a7      c2

1 Input Not Gate Location: R23C53A
  887266        f0      f8      f8      f0
  887693        95      5b      5b      95
  887694        71      14      14      71
  887699        f0      f8      f0      f8
  888126        68      a6      68      a6
  888127        a7      c2      a7      c2
```

Figure 19. Pins from Groups 5 and 6 showing the differences in changing indices.

Slew Rate and Drive.

Slew rate and drive were two other configuration options that were reverse engineered for the IOBs. The slew rate is the maximum voltage change per unit time in a node of a circuit. Each bidirectional or output pin can have a slew rate of either fast or slow. Input pins are always set to a slew rate of fast. Fast corresponds to high-speed performance while slow corresponds to low-noise performance. The drive attribute is applicable to all output and bidirectional pins and specifies the strength of the output signal in milliamps (mAs). Table 2 shows the different drive strengths available at each supply voltage. Additionally, not all drive levels are available for each Input/Output type. The I/O type used for all of the bitstreams was LVCMOS25 which did not allow a drive strength of 2 mA.

Initially, the slew rate and drive configuration options were approached with the same methodology that was used for pullmode. Generate bitstreams with all possible values of the single configuration option for each pin and then analyze. However, after generating bitstreams for both of these options independently it became clear that these options could affect each other. For example, after generating and analyzing the differences between fast and slow slew rate for all pins with a drive level of 12 mA, the same pins were analyzed at a drive level of 8 mA. It was impossible to discern the slew rate of the 8 mA pins using the information from the 12 mA drive level pins. Because of this, bitstreams were synthesized for each drive level for each slew rate and then analyzed together. After the bitstreams were generated they were compared against each other for each pin and then their binary was printed at those locations in order to visually recognize patterns between the bitstreams.

Once again the pins were analyzed in groups that corresponded to the groups they were split into during the pullmode analysis.

Table 2. Valid Drive Strengths

Drive Strength (mA)	VCCIO 1.2V	VCCIO 1.5V	VCCIO 1.8V	VCCIO 2.5V	VCCIO 3.3V
2	X				
4	X	X	X	X	
8	X	X	X	X	X
12	X	X	X	X	
16		X	X	X	X
20					X

Input/Output.

Out of the 295 pins only the 16 in Groups 2 and 3 are input only. All other pins can be set as either input, output, or bidirectional. The configuration option for these pins was first hinted at when analyzing the pullmode results. When examining Figure 29 there were clear patterns in the first four bitstreams where pin D19 was set

as an input pin and D18 was set as an output pin, and the second four bitstreams where pin D19 was set as an output pin and D18 was set as an input pin. That figure seemed to suggest that a `1101` in the beginning of a pin's section in the bitstream indicated an input pin and that a `1101` closer to the middle of a pins section in the bitstream indicated an output pin. To explore this hypothesis the previous bitstreams that had the pins set as inputs and outputs were compared to each other. Additional bitstreams were generated with the pins set as bidirectional and then compared to both the input and output bitstreams. The results of these comparisons can be found in the analysis section, 4.1.

3.5 Configurable Logic Blocks

The CLBs were the next item reverse engineered after the IOBs. The CLBs proved more difficult than IOBs simply because there are many more CLBs than IOBs and the LPF modification cannot be used to modify configuration options in the same way that was done previously. This is because the LUTs in the CLBs are configured based on the HDL when it is synthesized. The LPF is not used until the *map* and *place and route* steps in the bitstream generation process and therefore is not even considered until after the LUTs have been configured.

In order to overcome this issue, Lattice primitives were used along with HDL attributes. Each of the Lattice FPGAs have a library of primitives that are supported by that device. In the case of the ECP3, the LUT4 primitive was used so that the LUT could be directly initialized to the desired configuration value. The HDL attributes were used to set other attributes such as location instead of modifying the LPF simply to avoid having to change two different files. The LUTs are initialized based on what the output to their desired truth table is. Figure 20 shows an example of this. In order to get a LUT that would produce the outputs in the truth table shown, the

initialization value needs to be `0xF444`. When the synthesis process is translating HDL code into the bitstream, it replaces the logic in the design with LUTs that are initialized to produce the same outputs. Based on this information, the hypothesis was that the initialization information appeared somewhere in the bitstream. If this were the case, to understand the digital logic being implemented in the CLB one must simply figure out what the LUTs were initialized to and then remake the truth table.

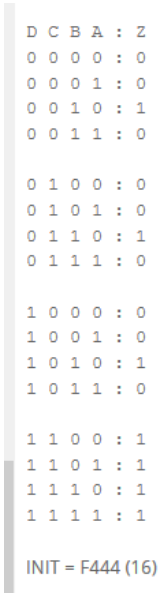


Figure 20. Look Up Table initialization value based on the desired truth table.

Single Look Up Table Reversal.

The process to reverse engineer the configuration of a LUT was to create a set of bitstreams that had a variety of configuration values for the same LUT. These bitstreams were then compared to each other in order to identify which indices were responsible for the configuration information. The bitstreams were then visually compared to each other at those indices to reveal how the configuration information was encoded within the bitstream. TCL scripts were again used to generate the

bitstreams.

Given that each LUT has a 16 bit configuration value, that 16 bit value was assumed to be stored somewhere within the bitstream. Therefore, at least 16 bitstreams would need to be generated for each LUT in order to locate the indices responsible. However, if other indices were also changing as a result of modifying the configuration value, additional bitstreams could possibly be necessary to identify the correct configuration indices. Therefore, 61 bitstreams were initially generated for the LUT. `0x0000` through `0x000F`, `0x0010` through `0x00F0`, `0x0100` through `0x0F00`, and `0x1000` through `0xF000`. This resulted in sufficient information to recognize how the configuration was stored in the bitstream.

Confirming The Mask Is Correct.

After a LUT was analyzed the result was a mask, or location of the bits specifying the encoding of the LUT configuration value. However, a confirmation was needed to ensure the information was correct and useful for understanding a bitstream. To do this, a number of bitstreams were synthesized with each bitstream containing different combinational logic located in the LUT of interest. Those bitstreams would then be used to recover the truth table for what that LUT was initialized to. If the recovered truth table and the specified logic agreed, the mask would be confirmed. Additionally, the logic would be specified using HDL operators instead of initializing the primitives directly. Using HDL operators would ensure that the initialization value of the LUT was generated by the synthesis engine translating the design. A number of different designs were created using simple logic such as a 3 input AND gate, 4 input OR gate, or a design that would combine various logic gates but was sure to fit in a single LUT. The design was then placed in the correct CLB and LUT using LPF or HDL constraints and analyzed.

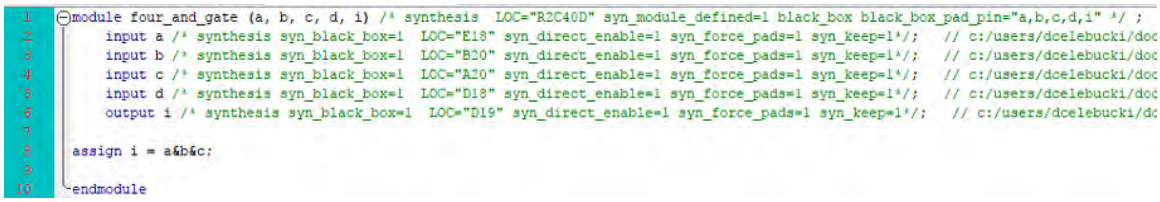


Figure 21. HDL used to generate a three input AND gate.

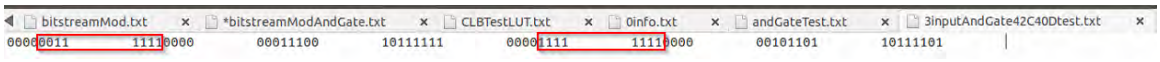


Figure 22. Bitstream values at R2C40D Look Up Table 1 indices for a three input AND gate.

This process is detailed for both a 3-input AND gate, as well as a more complicated design of $AB + C\bar{D}$. Figure 21 shows the HDL used to generate the three input AND gate and Figure 22 shows the values of the bitstream at the indices that are associated with LUT 1 in CLB R2C40D. The boxed red values correspond to the bits responsible for the configuration of the LUT. In this case there are `0's` in both the 16,384's and 32,768's place resulting in an initialization hex value of `0xC000`. If this value is used to derive the truth table for this LUT the outputs are `1` for input `1110` and `1111` which is exactly what is expected with a 3-input AND gate. All three of the inputs need to be high and then the fourth input does not have any effect on the function.

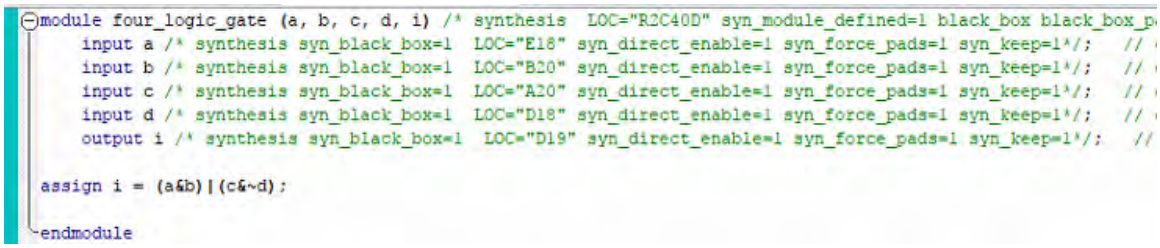


Figure 23. HDL used to generate a more complicated logic function.

The second example shows a more complicated logic function. Figure 23 shows the HDL used to generate the bitstream for the $AB + C\bar{D}$ while Figure 24 shows the synthesized bitstream at the indices that are associated with LUT 1 of CLB R2C40D.

Figure 24. Bitstream values at R2C40D Look Up Table 1 indices for a more complicated logic function.

When examining the bitstream it is more difficult to see if the derived mask is correct. When observing the bitstream and comparing it to the mask shown in Figure 46 a configuration hex value of `0x8f88` is derived. The digital logic function can then be reconstructed using the truth table that corresponds with the configuration hex value shown in Table 3.

Table 3. Derived Truth Table

W	X	Y	Z	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

This yields a logic function of:

$$\bar{W}\bar{X}YZ + \bar{W}XYZ + W\bar{X}\bar{Y}\bar{Z} + W\bar{X}\bar{Y}Z + W\bar{X}YZ + WXYZ$$

This function can then be further reduced to:

$$W\bar{X} + YZ$$

Although this isn't in the same form as the HDL one must remember that the synthesis process has control over how the inputs are routed to the LUT. Since the inputs can be routed in any order to the CLB we can replace the variables in our logic function. If W is replaced with C, X with D, Y with A and Z with B the logic function becomes $AB + C\bar{D}$ which was the same as the HDL code. This result shows that the function in the CLB is logically equivalent but the inputs were not necessarily routed in the same way. This finding means that the mask can be used to correctly predict the logic within a LUT. Even though the routing cannot be inferred from this process, it is still possible to understand the logic function that a LUT is replicating based solely on the bitstream.

3.6 Bitstream Modification Attack

A bitstream modification attack was performed using the information gained from reverse engineering how the configuration information for a LUT was stored. The goal was to simulate an attack where an adversary intercepts a bitstream en route from the IP designer and the target system. The adversary modifies the bitstream which is then loaded onto the target system. The process consisted of developing a simple digital circuit that was synthesized into a bitstream. The bitstream was loaded onto the target system to confirm that it was working correctly. The design was then modified by changing the bitstream to produce a new desired behavior and the modified bitstream was loaded onto the target system. If the new functionality

could be observed after the modified bitstream was loaded, the attack would be deemed successful which would also validate that bits identified as controlling the LUTs were correct.

Experiment Design.

The initial logic function was a simple 4-input OR gate. This OR gate was implemented using HDL operators instead of configuring the LUT directly to ensure that this was similar to the actual process an IP design would go through. Figure 25 shows the HDL used to generate the 4-input OR gate while Figure 26 shows the LPF used to locate the design into the correct LUT.

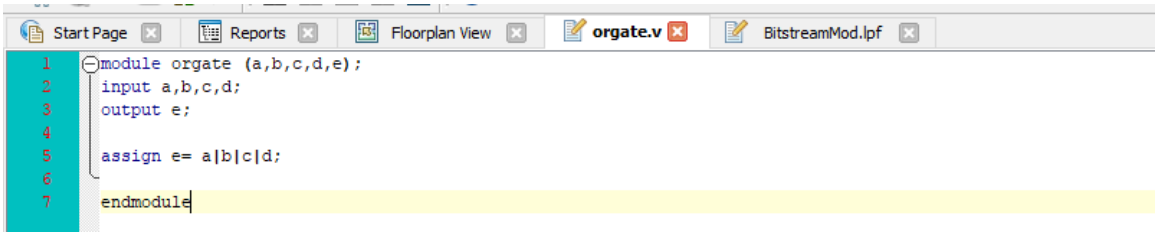


Figure 25. HDL used to generate the OR gate for the bitstream modification.

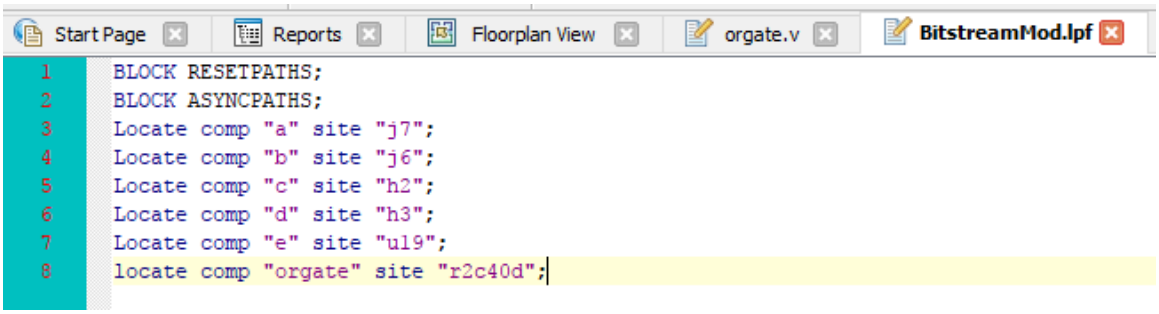


Figure 26. LPF constraints used to generate the OR gate for the bitstream modification.

The inputs were connected to 4 dip switches using LPF constraints and then the output was connected to a Light Emitting Diode (LED). After confirming that the design was functioning correctly on the target system, the bitstream was modified to

implement a 4-input AND gate using the information gained through reverse engineering the LUTs. Figure 27 shows the values that were replaced in the bitstream in order to make this change in the LUT in CLB R2C40D. The design was then loaded onto the target system where the change in functionality was observed and verified.

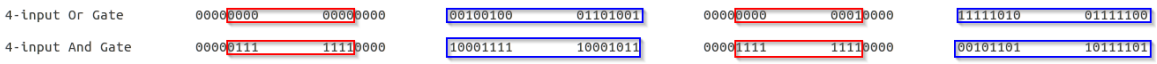


Figure 27. Indices that were modified to transform an OR gate into an AND gate.

3.7 Bitstream Parser

A bitstream parser was created in order to verify that the information assumed about the IOBs was correct and that a tool could be created to help protect FPGAs by inspecting bitstreams prior to loading them onto the FPGA.

Design and Testing.

The bitstream parser was created in python and takes a single bitstream as an input. The bitstream is transformed into a string of `1's` and `0's` that is then queried at various points in order to interpret what pins are being utilized and their configuration settings. Listing A.4 shows an excerpt from the script that provides the logic for parsing information about pin B4 from the bitstream. A number of IF statements are used to query different sections of the bitstream and then output information about the bitstream. The reasoning for the IF statements instead of a more advanced parser using tokenization was based on how the bitstream file is interpreted by the hardware. The bitstream configures options by directly setting selector values for multiplexers and is not interpreted in more sophisticated ways. Because of this, simply checking for combinations of different `1` and `0` patterns at different locations is sufficient for a parser.

The parser is able to parse information for 139 of the pins. This includes all pins in Groups 1 through 4 as well as pin C21 from Group 5. Pins in Groups 1 through 4 all followed very similar patterns and were rather easy to parse. It was much more difficult to narrow down the location of bits controlling the configuration options for pins in Groups 5 and 6 however. The increased difficulty is because these pins did not follow the same patterns as pins in Groups 1 through 4 and had their information spread throughout the bitstream in locations that could be thousands of indices apart. Due to time restrictions only a single pin was entered into the parser in order to show that it is possible to include them in the parser. Although the parser was partially tested as it was developed, after it was completed it was tested further to ensure that it would still parse IOBs and their configuration options even if there were many IOBs that all interacted together. In order to test this the MIL-STD-1553 encoder was used with all pins set to pins that were included in the parser. Their pullmodes, slew rates, and drive levels were picked randomly and then a bitstream was generated. Figure 28 shows the pins chosen and their configuration options. Pin C21 was purposely included to show that the parser would also work on pins from Group 5 and 6 even though all of them were not included. After the bitstream was generated it was input to the parser and the results were output to a text file. Results from this exercise can be found in section 4.4.

After the bitstreams for the IOBs and CLBs were generated, they were then analyzed in order to find the locations in the bitstream mapping to the various configuration options for each. The IOBs were analyzed in order to locate the bits that correspond to the pullmode, slew rate, and drive level configuration options as well as which bits determine whether a pin is an input or output pin. The CLBs were analyzed in order to find the bits corresponding to the initialization values of the LUTs. The information from both of these exercises was used to create the bitstream

Clock	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
enc_clk	N/A	C12	1	N/A	LVCMOS25	OFF	UP	NA	FAST
rst_n	N/A	AA19	3	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_csw	N/A	K1	7	N/A	LVCMOS25	OFF	KEEPER	NA	FAST
tx_dw	N/A	V20	3	N/A	LVCMOS25	OFF	DOWN	NA	FAST
tx_dword[0]	N/A	N17	2	N/A	LVCMOS25	OFF	NONE	NA	FAST
tx_dword[1]	N/A	R17	3	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[2]	N/A	B4	0	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[3]	N/A	T17	3	N/A	LVCMOS25	OFF	DOWN	NA	FAST
tx_dword[4]	N/A	T2	6	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[5]	N/A	T16	3	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[6]	N/A	T15	3	N/A	LVCMOS25	OFF	KEEPER	NA	FAST
tx_dword[7]	N/A	V17	3	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[8]	N/A	G9	0	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[9]	N/A	U15	3	N/A	LVCMOS25	OFF	NONE	NA	FAST
tx_dword[10]	N/A	B6	0	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[11]	N/A	T8	6	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[12]	N/A	A2	0	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[13]	N/A	W6	6	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[14]	N/A	V21	3	N/A	LVCMOS25	OFF	UP	NA	FAST
tx_dword[15]	N/A	D5	0	N/A	LVCMOS25	OFF	UP	NA	FAST
Output	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
tx_busy	N/A	Y17	3	N/A	LVCMOS25	OFF	DOWN	16	FAST
tx_data	N/A	C21	8	N/A	LVCMOS25	OFF	KEEPER	4	FAST
tx_dval	N/A	W17	3	N/A	LVCMOS25	OFF	NONE	8	SLOW

Figure 28. Pins and their configuration options used in the bitstream parser test.

parser and execute the bitstream modification attack.

IV. Results and Analysis

This chapter describes the results and accompanying analysis for the reverse engineering efforts, bitstream parser, and bitstream modification attack.

4.1 Input/Output Blocks

This section describes the results and analysis of the reverse engineering effort for the Input/Output Blocks (IOBs).

Pullmode.

The bitstreams related to the pullmode configuration option were analyzed in groups that corresponded to shared indices of change between bitstreams.

Group 1.

Group 1 consisted of 86 pins with changes at indices 476 or 477. The pins were sorted based on the smallest index of change found during their comparisons and the binary from index 0 to 477 for each of the bitstreams related to those pins when set as inputs was printed to the same file. Since each pin had four corresponding bitstreams the bitstreams would be printed in the same order for each pin (up, down, keeper, none) and then a blank line would be printed to signify the beginning of a new pin. Since the bitstreams were printed along the same indices and in order based on where changes were occurring in the bitstream, bits in the same column were in the same location in their respective bitstreams. Thus if there was a common pattern for how the pullmode was represented it would visually cascade through the resulting file.

Figure 29 shows a selection of this file at the first location of changes with the `1's` replaced with black squares and the `0's` replaced with white space for easier

viewing. The bits responsible for the pullmode option for each pin are boxed in red. These bits were determined responsible for the pullmode option because they were the only locations that change in a regular pattern for each pin in the group. For all pins in Group 1 pullmode is represented by 2 bits. Up is represented by `00`, down by `11`, keeper by `01`, and none by `10`. Additionally, Figure 29 explains the reason that the relationship between the pullmode values and the bitstream is not immediately apparent when looking at the hex results of the initial comparison like those shown in Figure 16. The pullmode bits for each of the pins shown in the image are 34 or 35 bits apart from each other. The bitstream does not abide by byte boundaries so the hex values representing the same pullmode option is different for different pins. Instead the bitstream uses its own boundaries denoted by `1's` after a certain number of bits. In Group 1 these boundaries are 34 or 35 bits apart. The bits within the boundaries correspond to options related to the same pin.

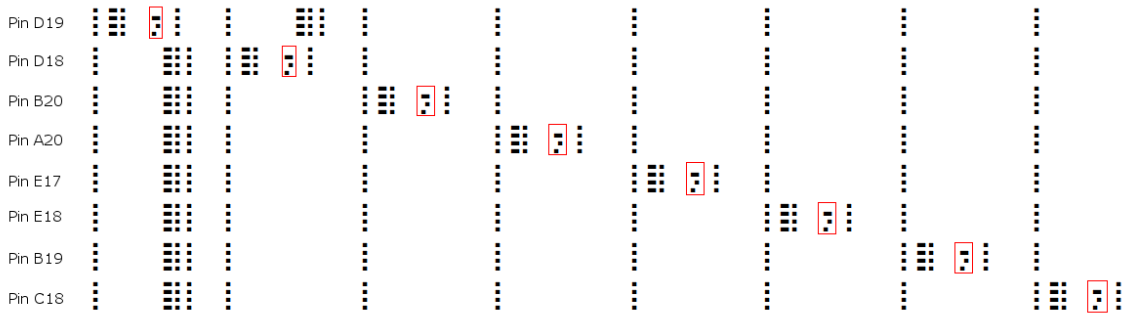


Figure 29. Selection of bitstreams related to the pullmode configuration of group 1 input pins with 1's replaced with black space and 0's replaced with white space.

After the locations of the bits responsible for each of the pins pullmodes were located, a file similar to the one in Figure 29 was created but instead used the bitstreams generated when the pins were set as outputs. This file was created to provide another data set to verify that the locations determined for the pullmode were correct. Figure 30 shows the same selection as Figure 29, but from the bitstreams related

to the pullmode of output pins. Although there are some bits that have changed, the pullmode bits are in the same locations and are expressed in the same pattern. This serves as further confirmation that these bits are responsible for the pullmode configuration option.

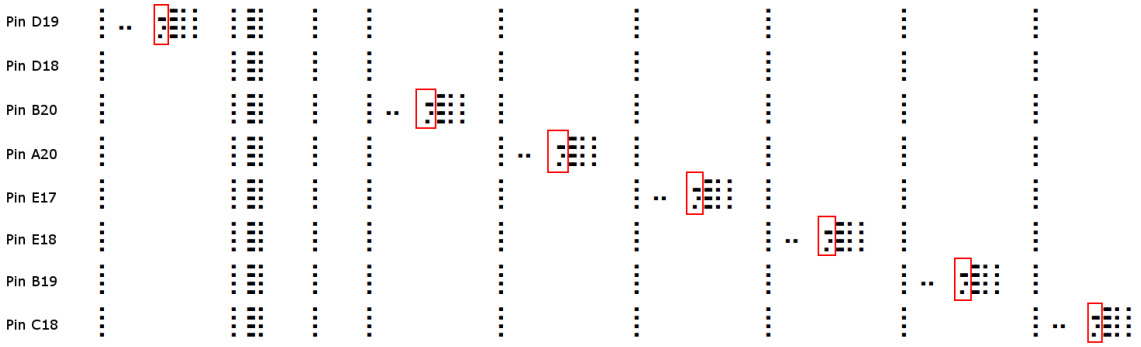


Figure 30. Selection of bitstreams related to the pullmode configuration of group 1 output pins with 1's replaced with black space and 0's replaced with white space.

Group 2.

Group 2 consisted of eight pins with changes at indices 440404 or 440405. All eight of these pins were input only which was confirmed by error messages when trying to generate bitstreams with these pins set as outputs. The binary for these pins was printed in much the same way except the indices ranged from 439980 to 440405. Figure 31 shows a selection of this file with the `1's` replaced with black space and the `0's` replaced with white space. The bitstreams are printed in the same order in that the first bitstream in any group of four relates to pullmode up, the second to pullmode down, the third to pullmode keeper, and the fourth to pullmode none. The middle bits of the file have been removed in order to show all eight of the pins in the same file, designated by the three ellipses in the middle of the image.

Once again, the pullmode is expressed using two bits however there is a slight change in the representation from the first four pins to the second four pins. The first

four follow the same pattern as the pins in Group 1. The second four pins though, switch the expression of keeper and none. Additionally the other bits expressed in each pin are mirrored in that they are on the other side of the pullmode bits.

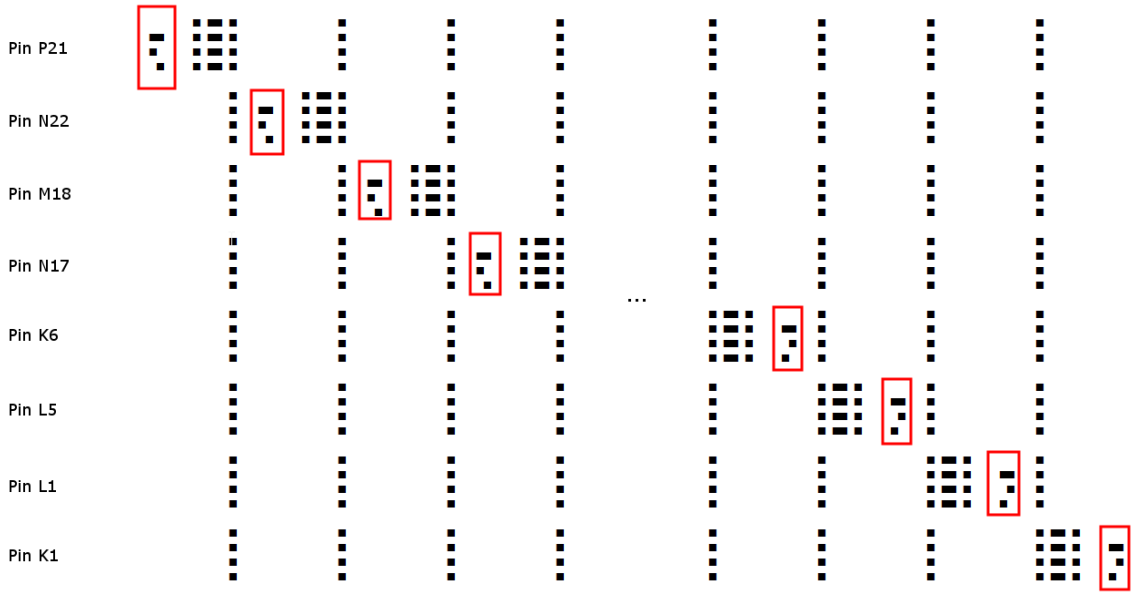


Figure 31. Selection of bitstreams related to the pullmode configuration of group 2 pins with 1's replaced with black space and 0's replaced with white space.

Group 3.

Group 3 consisted of eight pins with changes at indices 667296 or 667297. All eight of these pins were also input only pins similar to Group 2. The binary for these pins was printed from index 666870 to 667297. Figure 32 shows a selection of the printed binary with some of the middle bits removed in order to show the pullmode bits of all eight pins. The ellipses designates a removal of a portion of the file. These pins behave almost exactly like the pins in Group 2 with pullmode expressed with two bits and mirroring exhibited between the first four and second four pins.

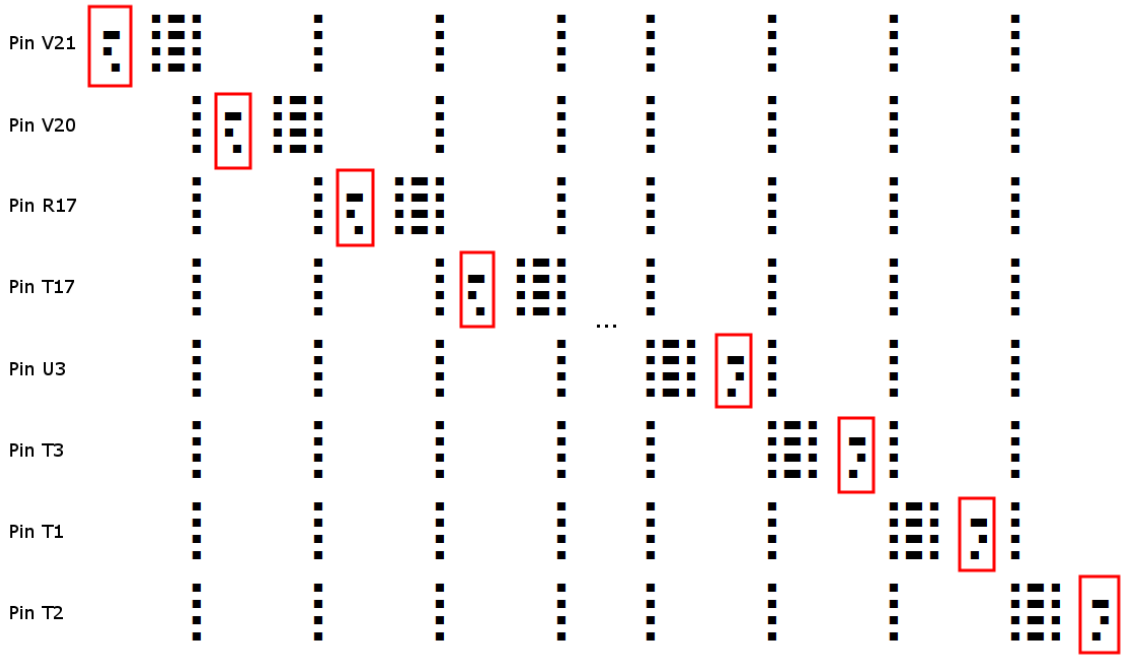


Figure 32. Selection of bitstreams related to the pullmode configuration of group 3 pins with 1's replaced with black space and 0's replaced with white space.

Group 4.

Group 4 consisted of 34 pins with changes at indices 895054 or 895055. These pins could be set as both input or output pins like Group 1. The binary for these pins was printed from index 894630 to 895055. Figure 33 shows a selection of the printed binary for these pins. Pins in Group 4 behaved very similarly to Group 1 in that they had 34 or 35 bits in between pullmode bits for subsequent pins and there was no mirroring exhibited.

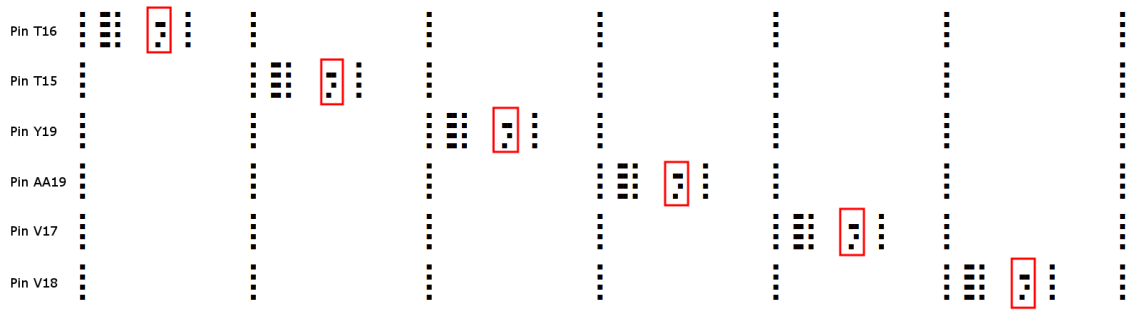


Figure 33. Selection of bitstreams related to the pullmode configuration of group 4 pins with 1's replaced with black space and 0's replaced with white space.

Group 5.

Group 5 was the first group of pins that did not follow the pattern of sharing indices. Instead this group was made up of any remaining pins that had an offset in the indices of 1, 2, 433, 434, 435. This accounted for 76 of the remaining pins. Since the indices of change for each pin were very far apart, only the indices directly responsible for each pin were printed. For example, the first 8 bits in the four bitstreams related to pin C21 were from byte 67597. This is very far removed from the first 8 bits for the first four bitstreams for pin D21 which come from byte 76690. This leads to a slightly different file than the file generated for the previous four groups. Instead of the changes cascading horizontally through the file, the pullmode bits responsible should manifest themselves as the only columns that do not change throughout the file. This is because if the pullmode bits for these pins follow the same pattern as the previous groups the bits responsible for pullmode should be at the same offset for each pin. Thus printing out only the indices that change for each pin will lead to the pullmode bits for each pin being in the same column. Figure 34 shows a portion of the resulting file. Once again individual bitstreams run horizontally across the image. A tabbed space between bits of the same bitstream indicates that bitstream is not

contiguous and the next bits after a tab do not follow directly after the previous bits. For example, the first eight bits printed in the four bitstreams related to pin C21 all come from byte 67597. The next 16 bits printed come from bytes 68024 and 68025. The following eight bits come from byte 68030 and the last 16 bits come from bytes 68457 and 68458. Additionally, since only the indices that changed for each pin were printed, bits that are in the same column no longer relate to the same location in their bitstreams. The boxed columns correspond to the bits relating to the pullmode for each pin. This was determined based on these are the only columns that stay the same for each pin. Assuming that these pins use a similar pattern of two bits of `00` for up, `11` for down, `10` for keeper, and `01` for none, and that they follow the same pattern of having the pullmode bits in a similar offset for each pin, these are the only locations that would fit.

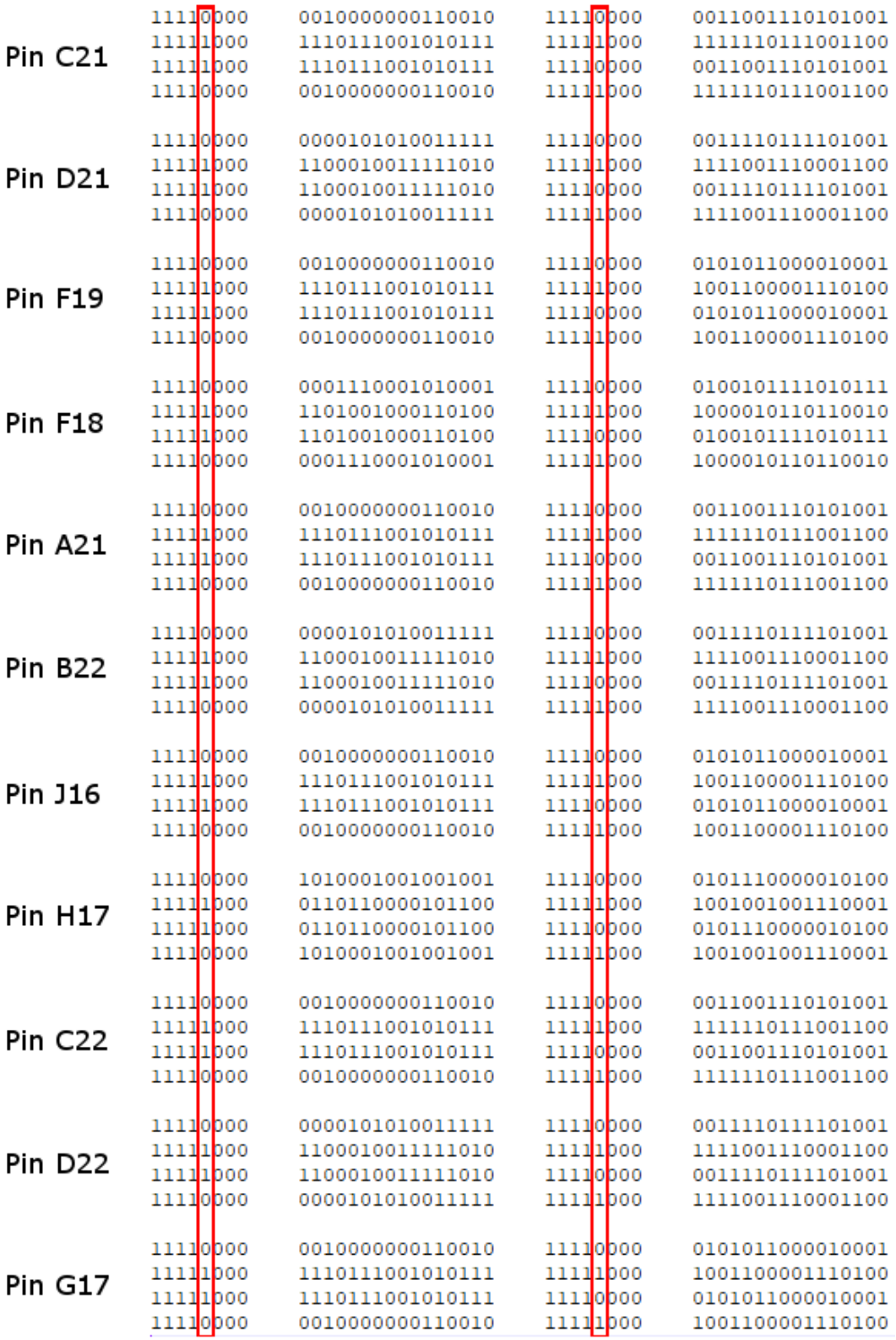


Figure 34. Selection of bitstreams related to the pullmode configuration of group 5 pins with 1's replaced with black space and 0's replaced with white space.

Group 6.

Group 6 was the second group of pins that did not follow the pattern of sharing indices. Like Group 5 this group contained pins that shared a common offset in the values of their indices of change. Any pins with an offset pattern of 427, 428, 433, 860, 861 were added to this group. This consisted of 81 pins. Similarly to Group 5, the binary of the bitstream files related to these pins was printed only at the indices of change. Figure 35 shows a portion of the resulting file. The file's organization is exactly the same as Group 5 so bits in the same column do not correspond to the same location in the bitstream. The red boxed areas correspond to the bits responsible for expressing the pullmode configuration for each of the pins.

Figure 35. Selection of bitstreams related to the pullmode configuration of group 6 pins with 1's replaced with black space and 0's replaced with white space.

Slew Rate and Drive.

The bitstreams related to the slew rate and drive level configuration options were analyzed in groups that corresponded to the groups created during the pullmode reverse engineering sections.

Group 1.

Group 1 consisted of the same 86 pins as Group 1 for the pullmode analysis. However, not all of the pins in Group 1 exhibited the same patterns when comparing drive and slew rate. In order to determine how slew rate and drive were expressed, the binary of the bitstreams from the bitstreams were printed from index 0 through 477. Figure 36 shows these bitstreams generated for pin D19 as well as the values of other generated bitstreams at the same locations for reference. The four bitstreams labeled "Input Pullmode" correspond to the four bitstreams generated expressing the pullmode options (up, down, keeper, none) when D19 was set as an input pin. "Output Pullmode" corresponds to the four bitstreams generated expressing the pullmode options when D19 was set as an output pin. "Bidirectional Pullmode" corresponds to the four bitstreams generated expressing the pullmode options when pin D19 was set as a bidirectional pin. "Just Slew Rate" shows the two bitstreams generated when only the slew rate option was changed. When comparing it to "Drive and Slew Rate" which has bitstreams generated for every drive and slew rate combination in the order 4 fast, 4 slow, 8 fast, 8 slow, 12 fast, 12 slow, 16 fast, 16 slow, 20 fast, 20 slow, it is easy to see that slew rate and drive affect the same bits. The red box encompasses the only bits that ever change when changing the drive or slew rate for this pin. This pattern is repeated for the first 8 pins and then the rest of the pins follow a slightly different pattern.

The other 78 pins follow a very similar pattern except for a one bit difference.

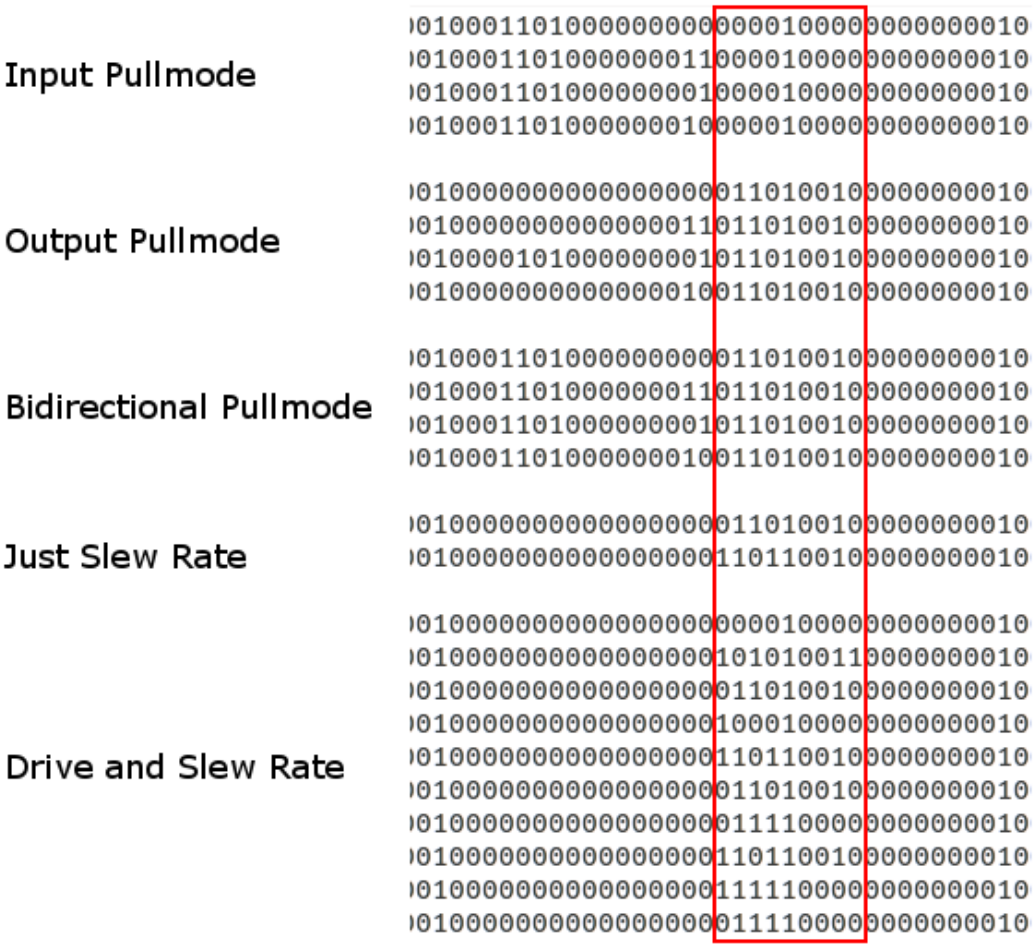


Figure 36. Comparison of bitstreams generated for pin D19.

Figure 37 shows the same bitstreams as Figure 36 but for Pin D17 instead of D19. The only difference is the seventh bit from the left inside the red enclosed area. This bit can change depending on other options such as the pullmode so it is not considered when trying to discern the slew rate or drive. The rest of pins in Group 1 follow this pattern.

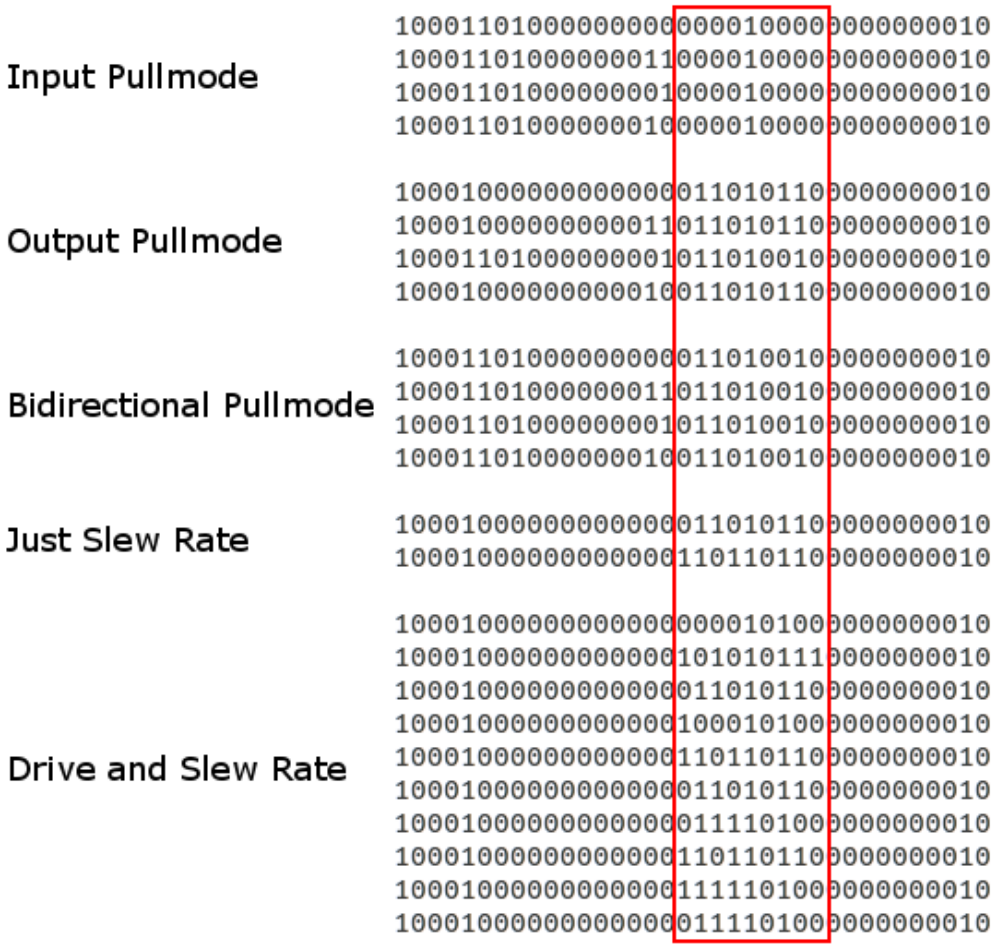


Figure 37. Comparison of bitstreams generated for pin D17.

Group 2 and Group 3.

Group 2 and Group 3 are input only pins and thus they have no drive level and their slew rates are always fast.

Group 4.

Group 4 consisted of the same 36 pins that were in Group 4 for the pullmode analysis. All pins in this group followed the same pattern that is demonstrated in Figure 38. Figure 38 has the same format as the previous figures with four bitstreams

for the four pullmode values for both pin T16 set as an input and an output followed by the 20 permutations of drive and slew rate. Once again the seventh bit is ignored when trying to discern the drive and slew rate of the pins.

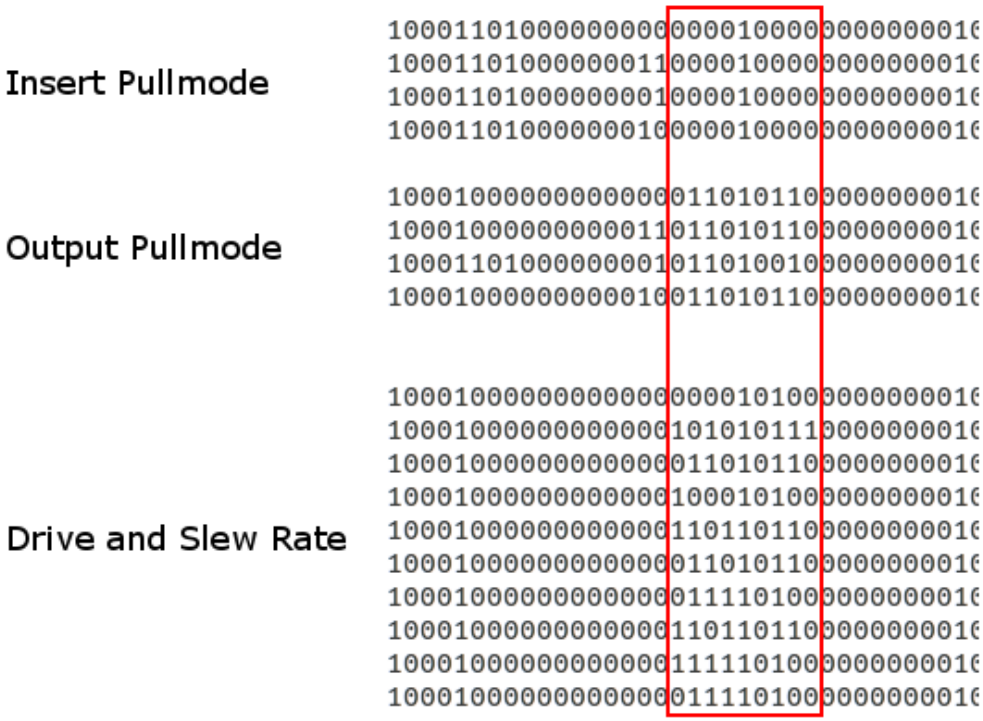


Figure 38. Comparison of bitstreams generated for pin T16.

Group 5 and Group 6.

Group 5 and 6 did not follow the same patterns as Groups 1 and 4 so discerning the slew rate and drive level were much more difficult. In the interest of time, only one pin was reversed and tested to show that it was possible to discern the slew rate and drive for pins in these groups.

Pin C21 which was the first pin in Group 5 during the pullmode analysis was picked to be the pin that was reversed. Once again the binary for the pin was printed at the indices that changed between different slew rates and drive levels. Figure 39 shows

the indices that changed when the drive level and slew rate were changed. The binary at these indices was then printed and compared for each of the 10 bitstreams. Using knowledge discovered during analysis of the Configurable Logic Blocks (CLBs) which showed checksum bits were represented by two adjacent bytes all possible checksum bits were eliminated and then the remaining indices were printed. This resulted in Figure 40. The boxed bits are the bits that are used to discern the slew rate and drive level of any bitstream using pin C21.

```
dan@ubuntu:~/Desktop/Bitstream Files/Scripts/test folder$ bitcompareV2 C214fast.bit C214slow.b
it C218fast.bit C218slow.bit C2112fast.bit C2112slow.bit C2116fast.bit C2116slow.bit C2120fast
.bit C2120slow.bit
   63700        f0      f8      f0      f0      f0      f0      f0      f0      f0      f0
   64127        59      97      59      59      59      59      59      59      59      59
   64128        0a      6f      0a      0a      0a      0a      0a      0a      0a      0a
   64133        f0      f8      f8      f0      f8      f8      f0      f8      f0      f0
   64560        0a      c4      c4      0a      c4      c4      0a      c4      0a      0a
   64561        9f      fa      fa      9f      fa      fa      9f      fa      9f      9f
   65865        f4      f4      f4      f4      fc      f4      fc      fc      fc      fc
   66292        bf      bf      bf      bf      71      bf      71      71      71      71
   66293        25      25      25      25      40      25      40      40      40      40
   66298        f0      f8      f8      f0      f0      f8      f8      f0      f8      f8
   66725        20      ee      ee      20      20      ee      ee      20      ee      ee
   66726        32      57      57      32      32      57      57      32      57      57
   66731        f0      f0      f8      f0      f8      f8      f8      f8      f8      f8
   67158        e6      e6      28      e6      28      28      28      28      28      28
   67159        43      43      26      43      26      26      26      26      26      26
   67164        f0      f8      f0      f8      f8      f0      f0      f8      f8      f0
   67591        9f      51      9f      51      51      9f      9f      51      51      9f
   67592        bb      de      bb      de      de      bb      bb      de      de      bb
```

Figure 39. Indices related to changing slew rate and drive for pin C21

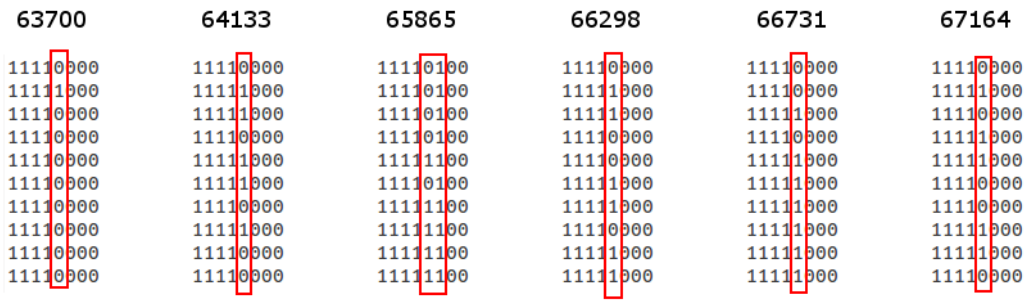


Figure 40. Comparison of bitstreams generated for pin C21.

Drive and Slew Rate Takeaways.

Unlike pullmode which used two bits for every pin in order to specify the value of that configuration option, drive and slew rate influenced the same bits and thus needed more bits in order to express the options. Additionally, certain drive levels and slew rate combinations were expressed using the same bit combinations in the bitstream. This suggests that those combinations are equivalent at the hardware level. In order to test this hypothesis and ensure that it was not a mistake in attributing which bits represented the drive level and slew rate, the bitstreams were compared. Figure 41 shows the result of the comparison between a bitstream with pin T15 using a 8 milliamp (mA) drive level and fast slew rate called `t158fast.bit` and another bitstream with pin T15 using a 12 mA drive level and a slow slew rate called `t1512slow.bit`. The comparison script produces a blank output meaning there are no differences between the two bitstreams. This shows that the hardware representations are most likely equivalent.

```
dan@ubuntu:~/Desktop/Bitstream Files/Scripts/test folder$ bitcompareV2 t158fast.
bit t1512slow.bit

dan@ubuntu:~/Desktop/Bitstream Files/Scripts/test folder$
```

Figure 41. Comparison of two bitstreams for pin T15 showing there was no difference between the 8 mA drive level with fast slew rate and the 12 mA drive level with slow slew rate.

Input/Output.

The analysis for the portions of the bitstream controlling whether a pin was an input or output was done using the bitstreams previously generated for the previous sections that were then compared to some additional bitstreams where the pins were set as bidirectional. The bitstreams were printed at the same indices that were printed when analyzing the pullmodes. Figure 42 shows an excerpt from the file generated

that reveals the input, output, and bidirectional configuration options for the pin D19. The first 4 bitstreams running horizontally from left to right show the four pullmode options (up, down, keeper, none) for pin D19 when set as an input. The next four show the same pullmode options but for pin D19 set as an output. The last four show the same pullmode options but for pin D19 set as bidirectional. The bits enclosed by the red, leftmost box are bits in the location that signify input. The bits enclosed by the blue, rightmost box signify output. Bitstreams that have both of these bits exercised high signify bidirectional.

Figure 42. Portions of bitstreams related to indices controlling input, output, and bidirectional options for pin D19.

This approach did not work for all pins however because there were certain pins

that did not have a single bit that only specified input. Instead these pins were checked to see if there were any options that were specified that would classify them as an output such as drive level and slew rate. If there were, they were then checked for bits that could possibly signify them as bidirectional. Figure 43 shows an example of one such pin. Figure 43 has its bitstreams organized in the same fashion in that the first four were all generated with pin D17 set as an input, the next four with D17 as an output, and the last four with D17 as bidirectional. The bits inside the rightmost blue rectangle were all checked and if they were `000010000` the pin was recognized as an input. If there was any other combination there the pin was either an output or bidirectional. From there the bits inside the leftmost rectangle were checked. If those bits were "1101" the pin is specified as bidirectional and if they were not the pin was specified as an output pin. This approach works well for most cases but can have difficulty discerning between bidirectional keeper and output keeper pins. However, this makes sense in some regards because of what is physically happening during those two cases. The keeper value for pullmode drives the pad to the last logic state. Since the logic drives the pad to the last logic state regardless of whether that state was an input or output the hardware will look the same for supporting both inputs and outputs.

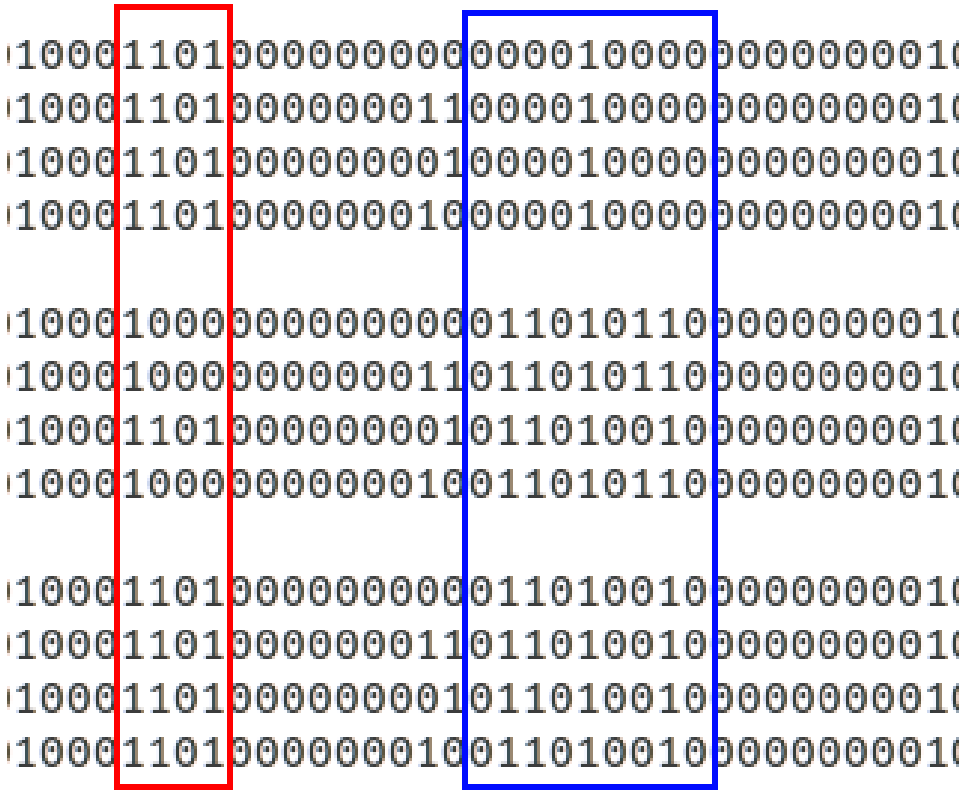


Figure 43. Portions of bitstreams related to indices controlling input, output, and bidirectional options for pin D17.

4.2 Configurable Logic Blocks

This section describes the results and analysis of the reverse engineering effort for the CLBs.

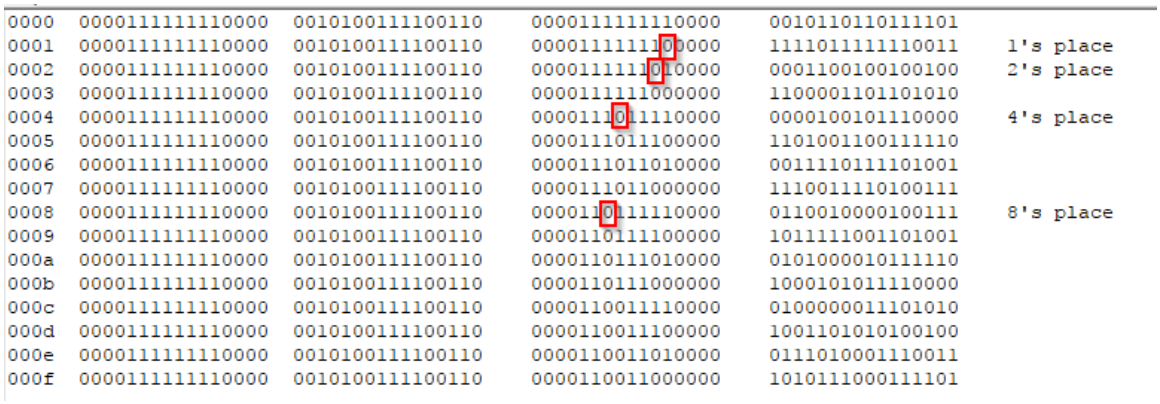
Single Look Up Table Reversal.

When comparing a set of bitstreams with different configuration options for the same Look-Up Table (LUT), there are between 6 and 8 bytes that change within the bitstream. The variation in the numbers of bytes changing has to do with the

bitstream not abiding by byte boundaries. There are 16 indices that correspond to the 16 bit initialization value that the LUT is configured to and 32 bits that act as some sort of checksum on those configuration bits. The configuration information however, is slightly encoded. If the initialization value of the LUT is considered as a 16 bit binary number, the indices are negated in that `1's` are replaced with `0's` and vice versa. Additionally, the 16 bits are not placed next to each other. They are spread across 2 to 4 bytes that can be 100's of indices away from each other in the bitstream. The bits responsible for encoding the configuration information were discerned by analyzing the differences between the individual bitstreams.

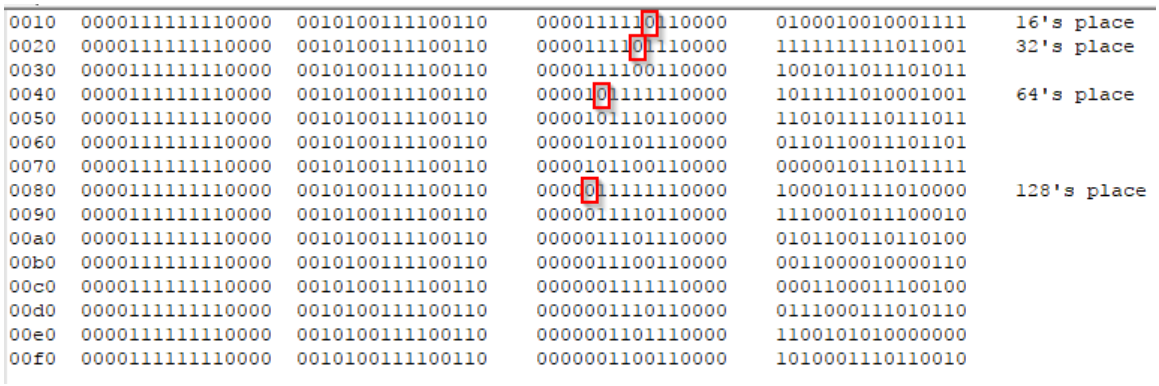
Figure 44 and Figure 45 show an example of this process. Each line in the files correspond to a bitstream for the same LUT that was initialized to different values. Figure 44 shows the first 16 bitstreams and Figure 45 shows the next 15 bitstreams all for the same LUT. The four values on the left show the hex representation of the 16 bit value the LUT was initialized to. The indices enclosed in red boxes correspond to the 1's, 2's, 4's, 8's, etc place locations of the corresponding hex value. This is observed by comparing which indices change from line to line. For example, the 1's place for the first hex value was confirmed by comparing the `0002` and `0003` initialization value lines. All of the indices in the last 16 indices do not change in a regular manner so they can be ignored. These would eventually be confirmed as a checksum on the configuration value while performing a bitstream modification attack. However, the change from `0002` to `0003`, has a `0` in the same location that the `0001` line has one. This can also be confirmed by comparing the `0004` and `0005` line, or any other line where the binary representation would only be changed in the 1's place. This process can be repeated for the rest of the configuration values. After the completion of this process many of the remaining bitstreams can be removed to show the "mask" or the indices that encode the 16 bit initialization value for a LUT. Figure 46 shows the

"mask" for LUT R2C40D. This process can then be completed on the rest of LUTs on the board to get the masks for each of them.



0000	0000111111110000	0010100111100110	0000111111110000	0010110110111101	
0001	0000111111110000	0010100111100110	0000111111100000	1111011111110011	1's place
0002	0000111111110000	0010100111100110	0000111111010000	0001100100100100	2's place
0003	0000111111110000	0010100111100110	0000111111000000	1100001101101010	
0004	0000111111110000	0010100111100110	0000111011110000	0000100101110000	4's place
0005	0000111111110000	0010100111100110	0000111011100000	1101001100111110	
0006	0000111111110000	0010100111100110	0000111011010000	0011110111101001	
0007	0000111111110000	0010100111100110	0000111011000000	1110011110100111	
0008	0000111111110000	0010100111100110	0000110111110000	0110010000100111	8's place
0009	0000111111110000	0010100111100110	0000110111100000	1011111001101001	
000a	0000111111110000	0010100111100110	0000110111010000	0101000010111110	
000b	0000111111110000	0010100111100110	0000110111000000	1000101011110000	
000c	0000111111110000	0010100111100110	0000110011110000	0100000011101010	
000d	0000111111110000	0010100111100110	0000110011100000	1001101010100100	
000e	0000111111110000	0010100111100110	0000110011010000	0111010001110011	
000f	0000111111110000	0010100111100110	0000110011000000	1010111000111101	

Figure 44. Bitstreams from a Look Up Table 1 in Configurable Logic Block R2C40D with different configuration values.



0010	0000111111110000	0010100111100110	0000111110110000	0100010010001111	16's place
0020	0000111111110000	0010100111100110	0000111101110000	1111111111011001	32's place
0030	0000111111110000	0010100111100110	0000111100110000	1001011011101011	
0040	0000111111110000	0010100111100110	0000101111110000	1011111010001001	64's place
0050	0000111111110000	0010100111100110	0000101110110000	1101011110111011	
0060	0000111111110000	0010100111100110	0000101101110000	0110110011101101	
0070	0000111111110000	0010100111100110	0000101100110000	0000010111011111	
0080	0000111111110000	0010100111100110	0000011111110000	1000101111010000	128's place
0090	0000111111110000	0010100111100110	0000011110110000	1110001011100010	
00a0	0000111111110000	0010100111100110	0000011101110000	0101100110110100	
00b0	0000111111110000	0010100111100110	0000011100110000	0011000010000110	
00c0	0000111111110000	0010100111100110	0000001111110000	0001100011100100	
00d0	0000111111110000	0010100111100110	0000001110110000	0111000111010110	
00e0	0000111111110000	0010100111100110	0000001101110000	1100101010000000	
00f0	0000111111110000	0010100111100110	0000001100110000	1010001110110010	

Figure 45. Continuation of bitstreams from a Look Up Table 1 in Configurable Logic Block R2C40D with different configuration values.

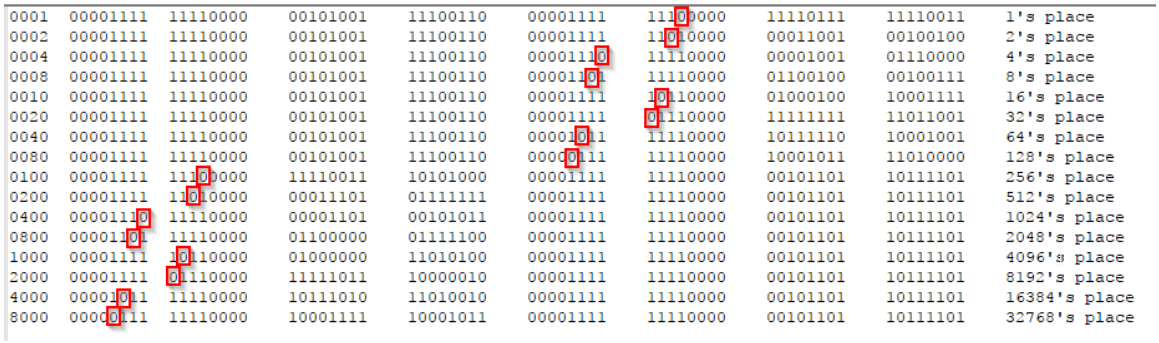


Figure 46. Mask for R2C40D Look Up Table 1.

4.3 Bitstream Modification Attack

Modification Results.

The original 4-input OR gate design was loaded onto the board and the correct functionality was observed. On this board, the Light Emitting Diodes (LEDs) are off when they are driven high. Figure 47 shows a subset of the states for the OR gate showing that the LED correctly lights up when the input is `0000` and the light is off for any other input. The red boxed areas correspond to the bits related to configuring the LUT while the indices boxed in blue correspond to the checksum bits. These bits were identified as some sort of checksum when trying to load the modified bitstream onto the target system. The programmer tool was capable of detecting modifications and would either return a file invalid report or xcf reader error. The xcf file is a configuration file used by the programmer which contains information about the device, data files targeted, and the operations to be performed [13]. However, when these bits were replaced with the checksum bits previously found when reverse engineering the mask, the programmer accepted the file. Figure 48 shows the target system after the modified bitstream was loaded. This shows that the circuit is displaying the intended behavior of an AND gate. The LED is correctly

on in every state except when the input is `1111`. This shows that the correct behavior is being exhibited after loading the modified bitstream onto the target system so the modification attack is a success.

Although the presence of the checksum bits increases the difficulty of the modification attack and using pre-synthesized checksums may not be feasible, the checksum can be defeated. For simple modifications the checksum can be brute forced since the indices checked by the various checksums are known. The other option is to reverse engineer the checksum algorithm. This can be done by running either the programmer or the Lattice Diamond software through a debugger to observe the operations responsible for calculating or checking the checksum. This method has been used in a very similar scenario where the encryption scheme on the Stratix II and Stratix III were uncovered making it highly applicable [22].

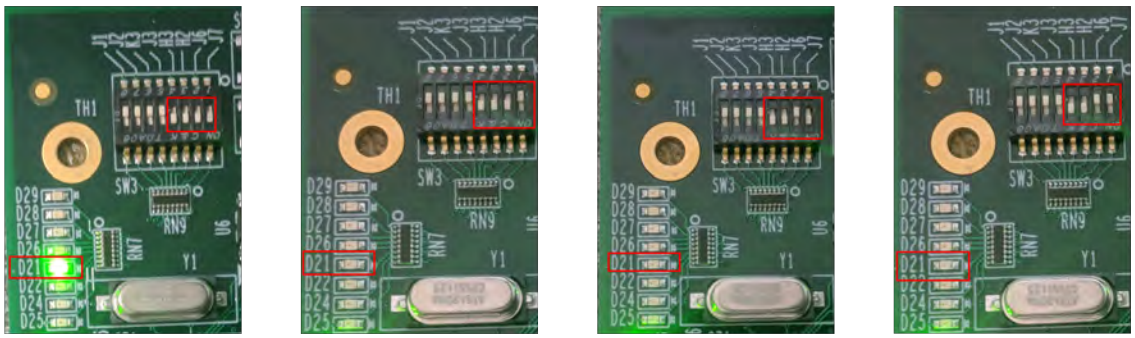


Figure 47. Various dip switch states showing the correct function of an OR gate.

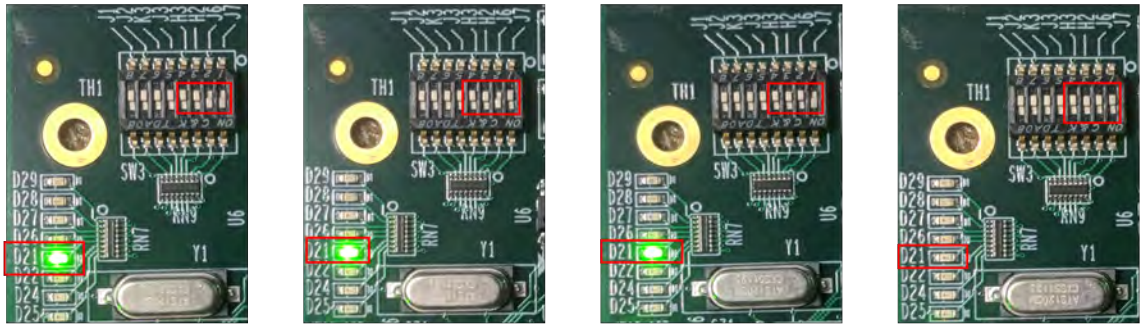


Figure 48. Various dip switch states showing the correct function of an AND gate after a bitstream modification attack.

4.4 Bitstream Parser

Figure 49 shows the result of the parser. Every pin was accurately detected and all configuration options were correctly identified. This shows that the information deduced about the IOBs in previous sections was correct and the bitstream parser can be used to identify components used in a design by strictly querying the bitstream.

The parser could also be expanded to include LUT information but that information was not included due to time constraints. Given that this parser can correctly identify components used in a design, it can be used to protect Field Programmable Gate Arrays (FPGAs). Consider the case where the owner of the FPGA is getting their designs from a third-party who is unwilling to disclose the source code. The owner can parse the design and save the output before uploading the bitstream. Then when an upgraded design is given to the owner they can parse the new bitstream and compare the outputs. In cases where additional input or output ports are used without prior reasoning given, the owner can discuss the changes with the third-party designer to ensure there are no covert channels or data leakage. Additionally, if the LUT information is added to the parser more data about the bitstreams including percentage of LUTs used, or even the logic in each of the LUTs could be provided to

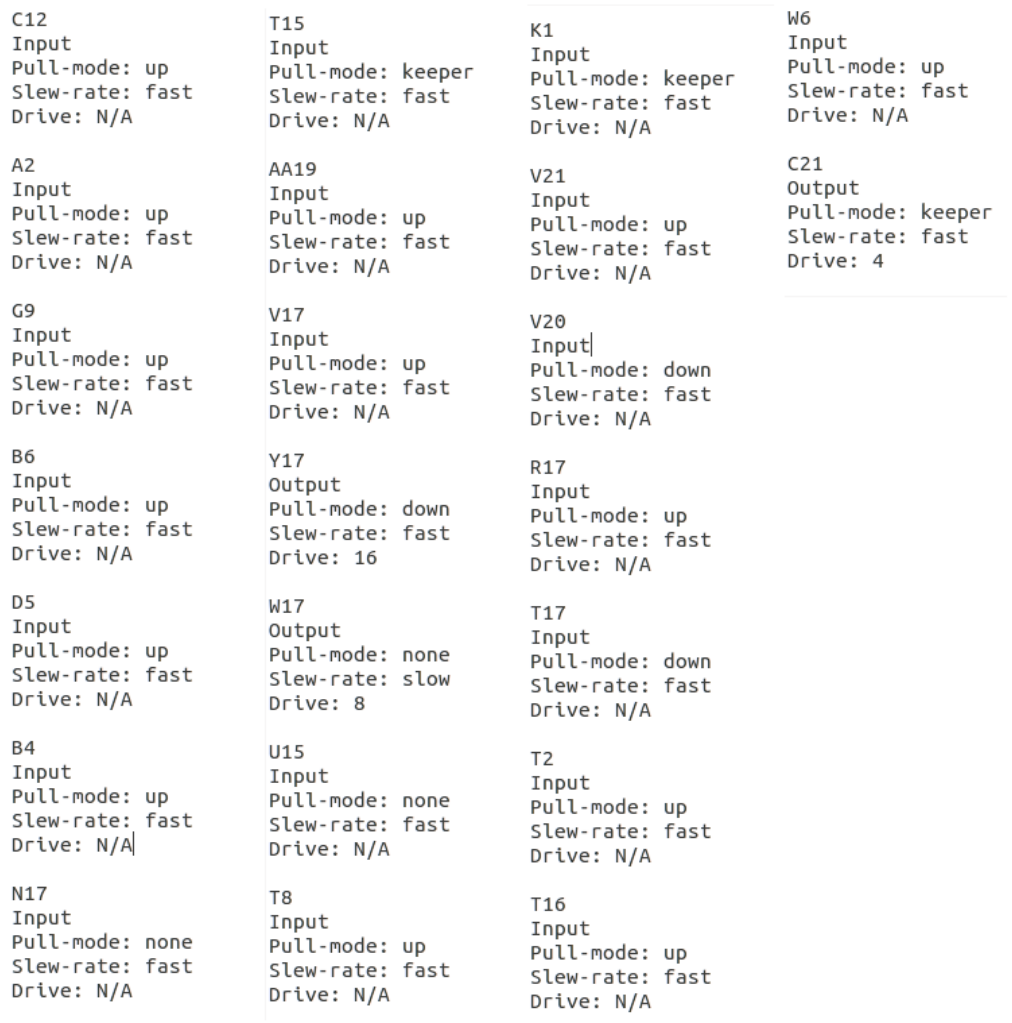


Figure 49. Results of testing the parser to parse a bitstream using pins with a variety of pullmodes, slew rates, and drive levels.

try to detect malicious hardware.

4.5 Comparison To Other Reverse Engineering Attempts

Although there have been a number of other attempts focused on reverse engineering FPGAs, this research has some key differences. This is the first research to the author's knowledge that focuses on a Lattice Semiconductor FPGA. Researchers in the past have mostly focused on Xilinx FPGAs [21, 2, 6], with less work focused

on Altera FPGAs [21, 22]. Additionally, there are key differences in the methods used to reverse engineer the bitstream formats. [21, 2, 6] all take advantage of the XDL and XDLrc files. The XDL file is a text file that is equivalent to the Native Circuit Description (NCD) file. The XDL and NCD files can be converted between each other and the XDL file gives a low-level description of the mapped and routed circuit internal details on the FPGA. The XDLrc file gives the overall internal detail information of the FPGA chip. This research does not make use of these file formats because they are specific to Xilinx FPGAs. Instead this research uses the Lattice Preference File (LPF) and Hardware Description Language (HDL) modifiers in order to modify low level details on the FPGA. [22] also makes use of a debugger in order to run the Quartus II program which allowed them to see exactly what actions the software was taking and the register values as the program was running. This approach was not used because it was faster to generate and analyze bitstreams in the way that was presented. However, their method could be used to solve the algorithm for the checksum bits found within the FPGA.

4.6 Overall Analysis

Various configuration options for IOBs and CLBs were mapped to their corresponding locations in the bitstream. For IOBs the pullmode options was successfully mapped for all 295 pins. Slew rate, drive level, and whether a pin was an input or output were mapped for all pins in Groups 1 through 4 and one pin in Group 5 was mapped to show that it was possible. For CLBs the location of bits specifying the initialization values for the LUTs was mapped for a few LUTs to show that the method was sound. However, in the interest of time not all LUTs were reverse engineered.

The information gained from the reverse engineering process was then used to create a bitstream parser that accurately parsed IOB information for all pins in Groups

1 through 4 and one pin in Group 5. Additionally, a bitstream modification attack was executed to show the masks for the LUTs were correct.

V. Conclusion

This chapter summarizes the goals (Section 5.1), conclusions (Section 5.2), contributions (Section 5.3) and future work (Section 5.4) of the research presented, and concludes with final recommendations (Section 5.5).

5.1 Motivation and Research Goals

Field Programmable Gate Arrays (FPGAs) are present in numerous industries including avionics, Industrial Control Systems (ICSs), and automobiles. These FPGAs are configured using a bitstream that has little to no inherent security measures. The presence of a sophisticated bitstream modification attack could affect nearly all FPGAs and could have drastic consequences both financially and physically. The goal of this research is to show that knowledge about a bitstream gained through reverse engineering can be used to create a parser to detect configuration options used in a design by strictly querying the bitstream possibly detecting malicious changes. Additionally, this knowledge can be used to execute a bitstream modification attack that interacts with the original design.

5.2 Conclusions

The research conclusions presented in each chapter are addressed here. First Chapter III presented the methodology used to reverse engineer portions of the LatticeECP3 LFE3-35EA-8FN484C bitstream. This process utilized no sophisticated third-party tools, instead relying solely on included functionalities of the Lattice Diamond tool and scripts written for comparison and analysis. Regarding the Input/Output Blocks (IOBs) the pullmode, slew rate, and drive level configuration options were successfully mapped to their counterparts in the bitstream. Additionally, portions

of the bitstream controlling whether a pin was utilized as well as if it was an input, output, or bidirectional were found. For the Configurable Logic Blocks (CLBs), the location of the bits representing the initialization values for a number of the Look-Up Tables (LUTs) were found. Additionally, these locations were tested to show that the logic implemented in a particular LUT could be found by noting the values at these bits and reconstructing a truth table. This process should be repeatable for any FPGA that utilizes the Lattice Diamond tool to synthesize its bitstreams.

In Chapter IV the results of the reverse engineering process were further shown to be reliable through the use of a parser that could output configuration information for a given bitstream and through the successful test of a bitstream modification attack which changed the initialization values in a LUT used in a Hardware Description Language (HDL) design. The parser was able to accurately detect the pins used and their configuration options for a variety of designs including a large MIL-STD-1553 encoder. The bitstream modification attack was a success and a 4-input OR gate that controlled the output to an Light Emitting Diode (LED) was transformed into a 4-input AND gate by strictly modifying the bitstream.

5.3 Contributions

This research presents another reverse engineering process, different from [21, 2, 6, 22], for a brand of FPGA not explored in past research. Additionally it shows that bitstreams can be analyzed prior to loading to detect differences from the specified design which could help to detect modifications due to an adversary. Finally, the success of the bitstream modification attack shows that modification attacks that affect the original design are possible and will still load onto the target FPGA.

5.4 Future Work

This research has significant potential for future work. The following sections detail additional work that could potentially improve detection of malicious hardware design or bitstream modification or perform more sophisticated attacks on FPGAs.

Switching Matrix.

Reverse engineering the switching matrix was not attempted due to time constraints. However, if the switching matrix could be fully or partially reverse engineered, the designs encoded in the bitstreams could be understood to a higher degree. Without the switching matrix it is impossible to know how the CLBs and IOBs are connected to implement the digital logic. This knowledge would allow for more detailed analysis of bitstreams to detect malicious hardware. However, fully reversing the switching matrix could also lead to adversaries using the knowledge to reverse engineer sensitive Intellectual Property (IP).

Further Parser Development.

The parser created could be further developed to include the information about the LUTs that was not included and additional configuration options. An increase in configuration options that the parser can detect could lead to higher chances of detecting malicious hardware that was included in the design or injected into the bitstream. Also the parser could be expanded to not only identify configuration information but to also analyze and possibly detect malicious hardware. This would require the switching matrix to be reverse engineered in order for the parser to be able to fully parse the digital logic.

More Sophisticated Bitstream Modification Attacks.

A more sophisticated bitstream modification attack could be attempted that interacts with a more complex design. The bitstream modification attack demonstrated was very simple and more of a proof of concept. Demonstration of an attack that changes a complex design possibly used in ICSs, or automobiles that either leaks data or results in dangerous behavior could be meaningful to show the validity of that threat.

Automated Reverse Engineering.

Although the analysis in the reverse engineering process for this research was largely manual it seems likely that the process could be automated. An automated system for reverse engineering bitstreams could be created using the information gained from this reverse engineering endeavor and then utilized on a different FPGA. This could result in parsers or bitstream modification attacks being developed for a number of different FPGAs instead of being confined to the FPGA used in this research.

5.5 Concluding Thoughts and Recommendations

This research shows the value of information gained from reverse engineering the format of a bitstream. The information can be used for both protection of an FPGA or to assist in an attack. Before more thorough bitstream parsing tools can be created or more sophisticated bitstream modification attacks can be attempted, several aspects of the future work presented must be addressed. This research presents a first step towards either of those goals on a system not previously explored in scientific literature. However, further development is required before an advanced malicious hardware detection system is created or an advanced bitstream modification attack

is achieved.

Appendix A. Scripts

Listing A.1. Example TCL script used to generate bitstreams

```
set globalcounter 1
#file_path is location of the project folders
set file_path "D:/Users/dance/Documents"
set drive_path "D:/Users/dance/Google Drive/AFIT/Thesis/Bitstream Files"
set output_folder "not_r2c73d_output"
#project_name is the specific folder
set project_name "pullmodetest1"
#name is name of the implementation before the number
set name pullmode
#val1 is starting implementation
set val1 0
#val2 is next implementation
set val2 585
set pattern NONE
set timestamp [clock format [clock seconds] -format {%Y%m%d%H%M%S}]
set firstRun 0
array set pinArray {
   1 D19
   2 A2
   3 A3
   4 A4
   # Rest of array omitted to save space

}
prj_project open "$file_path/$project_name/$project_name.ldf"
for {set index 147} {$index <= 295} {incr index} {
   set count 1
   for {set i 1} {$i <= 4} {incr i} {
      set systemTime [clock seconds]
      puts stderr "Progress globalcount:$globalcounter index:$index
          count:$count System time: [clock format $systemTime -format
          %H:%M:%S]"
      #set current project as active and export bitstream if you want
      if { $firstRun == 0 } {
      prj_impl active "$name$val1"
      set firstRun 1
      }

      #updating replace value
      if {$count == 1 } {
         set replace UP
      }
```

```
if {$count == 2 } {
   set replace DOWN
}

if {$count == 3} {
   set replace KEEPER
}

if {$count == 4} {
   set replace NONE
}



prj_impl clone "$name$val2" -dir "$name$val2" -impl "$name$val1"
    -copyRef
prj_impl active "$name$val2"

set filename
    "$file_path/$project_name/$name$val2/source/$project_name.lpf"
set temp $filename.new.$timestamp
set in [open $filename r]
set out [open $temp w]

#Loop through lines of the file
while {[gets $in line] != -1} {
   #do all changes here
   if {[regexp {LOCATE COMP "b" SITE} $line] == 1 } {
      puts $out "LOCATE COMP \"b\" SITE \"$pinArray([expr {$index +
          1}])\" ;"
   } elseif {[regexp {IOBUF PORT} $line] == 1 } {
      puts $out "IOBUF PORT \"b\" PULLMODE=$replace IO_TYPE=LVCMOS25
          ;"
   } else {
      puts $out $line
   }

}

#Close file access
close $in
close $out

#write the new LPF file
file rename -force $temp $filename
```

```
      #Synthesize the bitstream
   if {  [catch {
      prj_run Export -impl $name$val2 -task Bitgen

      #move bitstream to bitstream folder
      file rename -force
          "$file_path/$project_name/$name$val2/${project_name}_$name$val2.bit"
          "$drive_path/$output_folder/$pinArray([expr {$index +
          1}])$replace.bit"
   } err] } {
      puts stderr "Error encountered: $err $pinArray([expr {$index + 1}])"

   }
      incr globalcounter
      set val1 $val2
      incr val2
      incr count

   }
}
prj_project close
```

Listing A.2. Bitstream comparison program

```
//C program for comparing different bitstreams
//Note safety considerations have not been implemented

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>


#define SKIP_HEADER 1

int main(int argc, char *argv[])
{
//Get paths of files to compare

if(argc < 3){
   printf("Not enough args(3)\n");
   return 0;
}

   unsigned char headerEnd[] = {0x00, 0xff, 0xff, 0xff, 0xff, 0xbd, 0xb3,
       0x47, 0x00, 0x00, 0x00, 0xc2, 0x04, 0x80, 0x80, 0xff, 0xff, 0xff,
       0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
       0xff, 0xff, 0xff, 0x44};

//Read files into buffers
FILE *fp;
size_t sz;
size_t sz1;

   fp = fopen(argv[1],"rb");
   fseek(fp,0L, SEEK_END);
   sz = ftell(fp);
   rewind(fp);      //pretty sure this is unneeded
   fclose(fp);
   unsigned char (*buffer_array)[sz] = malloc(sizeof(double[argc-1][sz]));
   unsigned char tempBuffer[sz];
   int offset;
   int found = 0;
if(SKIP_HEADER == 1){
   for(int i = 1; i < argc; i++){
      fp = fopen(argv[i],"rb");
      fread(tempBuffer,sz,1,fp);
```

```
      for(int j = 0; j < sizeof(tempBuffer);j++){
         if(found == 1){break;}
         for(int k = 0; k < sizeof(headerEnd); k++){
            if(tempBuffer[j+k] == headerEnd[k]){
               if(k == sizeof(headerEnd -1)){
                  offset = j;
                  found = 1;
                  break;
               }

            }else{
               break;
               }
         }
      }
      found = 0;
      rewind(fp);
      fseek(fp,offset,SEEK_SET);
      sz1 = sz - offset;
      fread(buffer_array[(i-1)],sz1,1,fp);
      fclose(fp);
   }
}else{
   for(int i = 1; i < argc; i++){
      fp = fopen(argv[i],"rb");
      fread(buffer_array[(i-1)],sz,1,fp);
      fclose(fp);
   }
}

for(int i = 0; i<sz1; i++){
      for(int j = 1; j< argc - 1; j++){
         if(buffer_array[0][i] != buffer_array[j][i]){
            printf("%8d\t",i);
            for(int k = 0; k < argc - 1; k++){
            printf("%02x\t",buffer_array[k][i]);
            }
            printf("\n");
            break;
         }
      }
}

printf("\n");
return 0;
}
```

Listing A.3. Program for printing binary from bitstreams

```
//C program to take in bitstreams and output the binary of them
//May also include option to output hex aswell
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SKIP_HEADER 1
#define HEADER_LENGTH 334
#define BYTE_TO_BINARY_PATTERN "%c%c%c%c%c%c%c%c"
#define BYTE_TO_BINARY(byte) \
(byte & 0x80 ? '1' : '0'), \
(byte & 0x40 ? '1' : '0'), \
(byte & 0x20 ? '1' : '0'), \
(byte & 0x10 ? '1' : '0'), \
(byte & 0x08 ? '1' : '0'), \
(byte & 0x04 ? '1' : '0'), \
(byte & 0x02 ? '1' : '0'), \
(byte & 0x01 ? '1' : '0')

int main(int argc, char *argv[])
{
//Get paths of files to compare
int hexout = 0;
int binaryout = 0;
int commaCount = 0;
if(argc == 1){

   printf("1st argument either h for hex, b for binary, or B for both\n");
   printf("2nd argument is the indices list \n"
   "For example 0,440 would print from index 0 to index 440 \n"
   "0,440,500,700 would print from 0 to 440 and then 500 to 700\n"
   "There is no error checking on the indices so make sure you input them
      correctly\n"
   );
   printf("Arguments after this are the names of bitstreams\n");
   return 0;
}
//Using Strcmp which is dangerous so becareful with this if using in end
    tool
if(strcmp(argv[1],"h") == 0){
   hexout = 1;
   }else if(strcmp(argv[1],"b") == 0){
   binaryout = 1;
   }else if(strcmp(argv[1],"B") == 0){
   hexout = 1;
```

```
   binaryout = 1;
}

//Turn indices list argument into list of indices
char* indicesList;
indicesList = (char *) malloc(strlen(argv[2]) + 1);
strcpy(indicesList,argv[2]);
char* token;

token = strtok(indicesList, ",");
while(token != NULL){
   commaCount++;
   token = strtok(NULL,",");
}

int indicesArray[commaCount];
int counter = 0;

token = strtok(argv[2],",");
while(token != NULL){
   indicesArray[counter] = atoi(token);
   token = strtok(NULL,",");
   counter++;
}


   unsigned char headerEnd[] = {0x00, 0xff, 0xff, 0xff, 0xff, 0xbd, 0xb3,
       0x47, 0x00, 0x00, 0x00, 0xc2, 0x04, 0x80, 0x80, 0xff, 0xff, 0xff,
       0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
       0xff, 0xff, 0xff, 0x44};

//Read files into buffers
FILE *fp;
size_t sz;
size_t sz1;
   fp = fopen(argv[3],"rb");
   fseek(fp,0L, SEEK_END);
   sz = ftell(fp);
   rewind(fp);      //pretty sure this is unneeded
   fclose(fp);
   unsigned char buffer_array[(argc - 3)][sz];
   int offset;
   int found = 0;
if(SKIP_HEADER == 1){
   for(int i = 3; i < argc; i++){
      fp = fopen(argv[i],"rb");
```

```
      unsigned char tempBuffer[sz];
      fread(tempBuffer,sz,1,fp);

      for(int j = 0; j < sizeof(tempBuffer);j++){
         if(found == 1){break;}
         for(int k = 0; k < sizeof(headerEnd); k++){
            if(tempBuffer[j+k] == headerEnd[k]){
               if(k == sizeof(headerEnd -1)){
                  offset = j;
                  found = 1;
                  break;
               }

            }else{
               break;
               }

         }
      }
      found = 0;
      rewind(fp);
      fseek(fp,offset,SEEK_SET);
      sz1 = sz - offset;
      fread(buffer_array[(i-3)],sz1,1,fp);
      fclose(fp);
   }

}else{
   for(int i = 3; i < argc; i++){
      fp = fopen(argv[i],"rb");
      fread(buffer_array[(i-3)],sz,1,fp);
      fclose(fp);
   }
}


int startIndex;
int endIndex;

if(hexout == 1){
// printf("outputing Hex\n");
   for(int j = 0; j< argc - 3; j++){
      for(int k = 0; k < (sizeof(indicesArray)/sizeof(indicesArray[0])/2);
          k++){
         startIndex = indicesArray[2*k];
         endIndex = indicesArray[2*k+1];
```

```
         if(endIndex > sz1){
            endIndex = sz1;
         }
         for(int i = startIndex; i<=endIndex; i++){
            printf("%02x",buffer_array[j][i]);
         }
         printf("\t");
      }
      printf("\n");
   }
}

if(binaryout == 1){
   //printf("outputing Binary\n");
   for(int j = 0; j< argc - 3; j++){
      for(int k = 0; k < (sizeof(indicesArray)/sizeof(indicesArray[0])/2);
          k++){
         startIndex = indicesArray[2*k];
         endIndex = indicesArray[2*k+1];
         if(endIndex > sz1){
            endIndex = sz1;
         }
         for(int i = startIndex; i<=endIndex; i++){
            printf(BYTE_TO_BINARY_PATTERN,BYTE_TO_BINARY(buffer_array[j][i]));
         }
         printf("\t");
      }
      printf("\n");
   }
}


}
```

Listing A.4. Excerpt from bitstream parsing script

```
#B4
input = 0
if bitstream[3719] == "1":
    #print(bitstream[464:472])
    print("B4")
    if bitstream[3715:3724] == "000010000":
            print("Input")
            input=1;
    else:
        if bitstream[3702:3707] == "1101":
            print("Bidirectional")
        else :
            print("Output")

    print("Pull-mode: ",end='')
    if bitstream[3713:3715] == "00":
        print("up")
    elif bitstream[3713:3715] == "11":
        print("down")
    elif bitstream[3713:3715] == "01":
        print("keeper")
    elif bitstream[3713:3715] == "10":
        print("none")

    if input == 1:
        print("Slew-rate: fast")
        print("Drive: N/A")
    else:
        if bitstream[3715:3721] == "101010" and bitstream[3722:3724] ==
            "11":
            print("Slew-rate: slow")
            print("Drive: 4")
        elif bitstream[3715:3721] == "000010" and bitstream[3722:3724] ==
            "00":
            print("Slew-rate: fast")
            print("Drive: 4")
        elif bitstream[3715:3721] == "100010" and bitstream[3722:3724] ==
            "00":
            print("Slew-rate: slow")
            print("Drive: 8")
        elif bitstream[3715:3721] == "110110" and bitstream[3722:3724] ==
            "10":
            print("Slew-rate: fast")
            print("Drive: 12")
        elif bitstream[3715:3721] == "011010" and bitstream[3722:3724] ==
```

```
            "10":
            print("Slew-rate: slow")
            print("Drive: 12")
        elif bitstream[3715:3721] == "011110" and bitstream[3722:3724] ==
            "10":
            print("Slew-rate: fast")
            print("Drive: 16")
        elif bitstream[3715:3721] == "111110" and bitstream[3722:3724] ==
            "00":
            print("Slew-rate: fast")
            print("Drive: 20")
        elif bitstream[3715:3721] == "011110" and bitstream[3722:3724] ==
            "00":
            print("Slew-rate: slow")
            print("Drive: 20")


    print()
```

Bibliography

1. Altera Corporation. *Configuring Cyclone II Devices*, 2007.

2. Florian Benz, André Seffrin, and Sorin A. Huss. Bil: A tool-chain for bitstream reverse-engineering. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 735–738, Aug 2012.

3. Rajat. S. Chakraborty, Indrasish. Saha, Ayan Palchaudhuri, and Gowtham. K. Naik. Hardware Trojan Insertion by Direct Modification of FPGA Configuration Bitstream. *IEEE Design Test*, 30(2):45–54, April 2013.

4. E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.

5. Mathias Lasser Clifford Wolf. IceStorm. `http://www.clifford.at/icestorm/`.

6. Zheng Ding, Qiang Wu, Yizhong Zhang, and Linjie Zhu. Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. *Microprocessors and Microsystems*, 37(3):299–312, 2013.

7. Saar Drimer. Volatile FPGA design security a survey. *University of Cambridge*, pages 1–51, 2008.

8. Saar Drimer. *Security for volatile FPGAs*. Phd dissertation, University of Cambridge, 2009.

9. Umer Farooq, Zied Marrakchi, and Habib Mehrez. FPGA Architectures: An Overview. In *Tree-based Heterogeneous FPGA Architectures*, pages 7–48. 2012.

10. Sezer Gören, Ozgur Ozkurt, Abdullah Yildiz, H. Fatih Ugurdag, Rajat S. Chakraborty, and Debdeep Mukhopadhyay. Partial bitstream protection for low-cost FPGAs with physical unclonable function, obfuscation, and dynamic partial self reconfiguration. *Computers and Electrical Engineering*, 39(2):386–397, 2013.

11. R. Karam, T. Hoque, S. Ray, M. Tehranipoor, and S. Bhunia. Robust bitstream protection in fpga-based systems through low-overhead obfuscation. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Nov 2016.

12. Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973.

13. Lattice Semiconductor Corporation. *Help for Lattice Diamond*, 2014.

14. Enno Lübbers. Configurable System-on-Chip: Xilinx EDK.

15. Sanchita Mal-Sarkar, Aswin Krishna, Anandaroop Ghosh, and Swarup Bhunia. Hardware trojan attacks in fpga devices: Threat analysis and effective counter measures. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, GLSVLSI '14, pages 287–292, New York, NY, USA, 2014. ACM.

16. J. Millen. 20 years of covert channel modeling and analysis. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*, pages 113–114, 1999.

17. Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of fpga bitstream encryption against power analysis attacks: Extracting keys from xilinx virtex-ii fpgas. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 111–124, New York, NY, USA, 2011. ACM.

18. Amir Moradi, Markus Kasper, and Christof Paar. On the portability of side-channel attacks - an analysis of the xilinx virtex 4 and virtex 5 bitstream encryption mechanism. 2011:391, 01 2011.

19. Kevin Morris. All is Not SRAM. *FPGA Journal*, pages 5–9, 2007.

20. National Instruments. Introduction to FPGA Hardware Concepts (FPGA Module), 2011.

21. Jean-Baptiste Note and Éric Rannaud. From the bitstream to the netlist. *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, page 264, 2008.

22. Pawel Swierczynski, Amir Moradi, David Oswald, and Christof Paar. Physical security evaluation of the bitstream encryption mechanism of altera stratix ii and stratix iii fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):34:1–34:23, December 2014.

23. Stephen M. Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, 103(3):318–331, March 2015.

24. Stephen M. Trimberger and Jason J. Moore. FPGA security: Motivations, features, and applications. *Proceedings of the IEEE*, 102(8):1248–1265, 2014.

25. Xilinx. 7 Series FPGAs Configuration [UG470 v1.12]. 470:1–180, 2017.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
22–03–2018	Master's Thesis	Sept 2016 — Mar 2018

4. TITLE AND SUBTITLE

Methods of Reverse Engineering a Bitstream for Field Programmable Gate Array Protection

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Celebucki, Daniel, J., 2d Lt, USAF

5d. PROJECT NUMBER

17G385

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-18-M-018

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
2241 Avionics Circle
WPAFB OH 45433-7765
Attn: Lt Col Patrick Sweeney
COMM 937-713-4252
Email: Patrick.Sweeney@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYWA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Field Programmable Gate Arrays (FPGAs) are found in numerous industries including consumer electronics, automotive, military and aerospace, and critical infrastructure. The ability to be reprogrammed as well as large computational power and relatively low price make them a good fit for low-volume applications that cannot justify the Non-Recurring Engineering (NRE) costs associated with producing Application-Specific Integrated Circuits (ASICs). FPGAs however, have seen a variety of security issues stemming from the fact that their configuration files are not inherently protected.
This research assesses the feasibility of reverse engineering the bitstream format for a previously unexplored FPGA, as well as the utilization of the knowledge gained during that process to create a bitstream parser and perform a bitstream modification attack. The reverse engineering process utilizes Tool Command Language (TCL) scripts to automate the modification of various configuration options and then synthesize the resulting bitstream. Various configuration options for Input/Output Blocks (IOBs) are mapped to their respective locations in the bitstream and the encoding format for the configuration of several Look-Up Tables (LUTs) is discovered.
This information is then utilized to create a bitstream parser that takes a bitstream as an input and outputs configuration information for IOBs. Additionally, a bitstream modification attack is performed that changes the original design logic by modifiying the bitstream directly to change the configuration values of a LUT. Both the parser and bitstream modification attack are shown to work validating the information gained through the reverse engineering process.

15. SUBJECT TERMS

FPGA, Reverse Engineering, Bitstream

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Scott Graham, AFIT/ENG
U	U	U	U	106	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, Scott.Graham@afit.edu