Maximizing the Effect of Redundancy on Fault Diversification Through Core Layout

Alec Kohlhoff and Kyle Scaplen

Worcester Polytechnic Institute

December 13, 2018

A Major Qualifying Project

Submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelors of Science

APPROVED:

Professor Andrew Clark
Project Advisor

Professor Robert Walls
Project Advisor

Contents

List of Figures

List of Tables

1 Introduction

Embedded devices play a large role in the way the world operates on a daily basis. Many individuals, companies, and organizations rely on these devices to successfully complete specific tasks for various applications. Their uses include communications, signal processing, computing, entertainment, sensor control systems, security, and many more. However, embedded systems are vulnerable to threats that may inhibit the device from completing its desired task.

The specific type of threat our research focuses on is electromagnetic fault injection (EMFI). EMFI is a physical attack on the embedded device. This means that the attacker is not gaining any access to the software used on the device, but is simply subjecting the device to conditions that cause the hardware to either give information to the attacker or act in an unexpected manner. In an EMFI threat, the attacker actively causes an elecromagnetic pulse at a high voltage for a short amount of time. This spike in voltage can induce a current in the chip, which may change logic values or disrupt the function of the processor.

When defending against this type of threat, redundancy is considered a possible countermeasure. The basic implementation of redundancy is to duplicate any critical sections of a device multiple times, and have each individual section execute in parallel. That way if one of the sections is subjected to an attack, the other sections will execute the correct behavior and the device can continue processing as intended. However, this type of redundancy fails when an attacker has the resources to duplicate their attack and fault all redundant sections at the same time. This means that every section may experience unexpected behavior.

Building on the basic implementation of redundancy, it may be possible to create redundant modules of a device that when faulted, their faulty outputs or behavior will never be the same. This is called *fault diversification.* This would improve the effectiveness of redundancy because if all faulty outputs match, it is difficult or impossible to determine if it is in fact a faulty output or if that behavior was the correct execution of the device. However, if the faulty outputs are guaranteed to never match across implementations of redundant modules, you can better defend against an EMFI threat.

To research fault diversification, we constructed different implementations of the ALU module from a soft-core processor. The different implementations had various hardware layouts within an FPGA. By placing hardware elements of the processor in different arrangements, it may cause faults that have different effects on the device.

To explore if fault diversification through core layout is successful, we completed the following tasks:

- deployed a soft-core processor by extracting the ALU module from the mor1kx soft-core processor and synthesizing the module in Vivado
- conducted simulations to model the behavior of various core layouts by instantiating the ALU module in a test fixture and controlling the clock signals input to different parts of the module to observe how different implementations may behave during a physical implementation
- analyzed simulation data to select core layouts for lab testing that were most likely to cause fault diversification between layouts

- performed physical EMFI lab testing on the selected core layouts using Vivado placement tools to create each layout and a pulse generator with an EM probe to actively cause faults in the soft-core processor

- analyzed lab test data to understand the behavior of various core layouts subjected to EMFI in a physical environment to determine if fault diversification was achieved

This paper outlines the background research conducted to better understand our goal, describes the methods we used to complete our objectives, explains how the simulations and lab tests were implemented, and presents the results from the research.

2 Background

This section provides context for the project by describing field-programmable gate arrays, soft-core processors, clock regions, faulting, and the current state of electromagnetic fault injection (EMFI) countermeasures.

2.1 Field-Programmable Gate Arrays

A field-programmable gate array (FPGA) is a re-programmable semiconductor device used to implement custom logic designs in hardware [1]. FPGAs are composed of three major elements: configurable logic blocks (CLBs), input/output blocks (IOBs), and programmable interconnects [2]. Figure 1 shows how the three parts are situated within an FPGA.

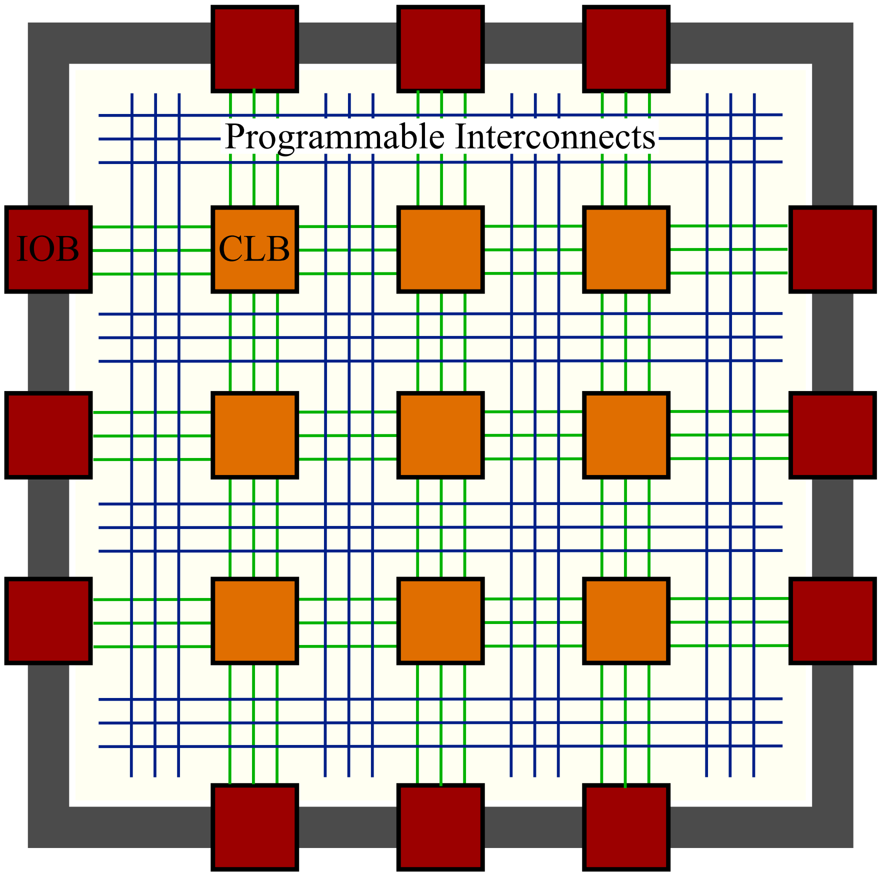

Figure 1: The elements of an FPGA.

The CLBs are responsible for implementing logic. The two building blocks of a CLB are RAM-based look-up tables (LUTs) and flip-flops. LUTs implement the user's custom logic by taking a number of inputs and producing an output in the form of a truth table. A simple digital logic element, the `AND` gate, can be represented by a truth table, which is shown as an example in table 1. Flip-flops are storage units, each holding a single bit of information. They keep the single bit in memory between clock edges. At a new clock edge, the flip-flop outputs the stored bit, receives a new bit from the input, and then stores that bit until the

Table 1: Truth table for an and gate.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

next clock edge. These units can be used individually or together in order to synchronize logic or save logic states [2].

The role of IOBs is to communicate with external devices. Any data or signal that the FPGA receives or transmits will ultimately pass through one of these blocks. Finally, CLBs and IOBs can be routed together in combination with each other using the programmable interconnects [2]. Utilizing these three main features, a developer can create a complex, customized design in programmable hardware using an FPGA.

2.2 Embedded Processors

Embedded devices perform specific tasks for various applications by joining both hardware and software components [3]. Many of these devices contain microprocessors, or integrated circuits with the functions of a central processing unit in a computer. Processors for embedded devices come in two forms: hard-core and soft-core. A processor is *hard-core* when it has a dedicated, physical implementation in the silicon [4]. In this case, the processor cannot be changed to better fit an application, but they are built to provide greater performance and efficiency. Likewise, peripherals are specific to the processor and the purpose of the device. In contrast, a *soft-core* processor implementation uses digital logic in an FPGA. Developers use a hardware description language (HDL) to describe the architecture and behavior of the processor [3].

By sacrificing performance and efficiency, soft-core processors are able to provide greater flexibility. When engineering for specific applications, the processor design is highly configurable and allows for modifications without replacing physical hardware on the board. For example, as project requirements are adjusted or added, engineers can make rapid adjustments to the processor design as needed rather than having to undergo a full redesign for a new hard-core processor [4]. Developers can quickly add or remove peripherals, such as a universal asynchronous receiver transmitter (UART), depending on the application. Designers can specify the exact location of each submodule on the silicon die. Also, one can specify a unique pin-out on the board by routing specific signals to a user defined location of the chip. These tools bring increased functionality and adaptivity to the engineering design process.

2.3 FPGA Clock Regions

When determining the core layout of an FPGA design, it is important to consider what clock regions are, how they are organized in an FPGA, and how they may affect an implementation. The general clocking architecture is set up with both global and regional clocking resources [5]. Global resources are accessible throughout the FPGA and regional resources are accessible only by a specific section of the FPGA. Clock regions are simply groupings of CLBs and other FPGA resources into smaller, rectangular sections. Across Xilinx 7 series devices, the number of clock regions ranges from 1 to 24 [5]. The lines that distribute the clock signal to parts of the FPGA form a "clock tree". The lines originate from a clock buffer and branch into different sections. A global clock tree allows the clocking of synchronous elements across the device. The global clock backbone runs down the entire length of the FPGA. A combination of global clock buffers, horizontal clock buffers, and regional clock buffers distribute the clock signal to separate regions [5]. Fig. 2 shows an overview of a generic clock architecture in an FPGA. As section 2.2 describes, a user can interact

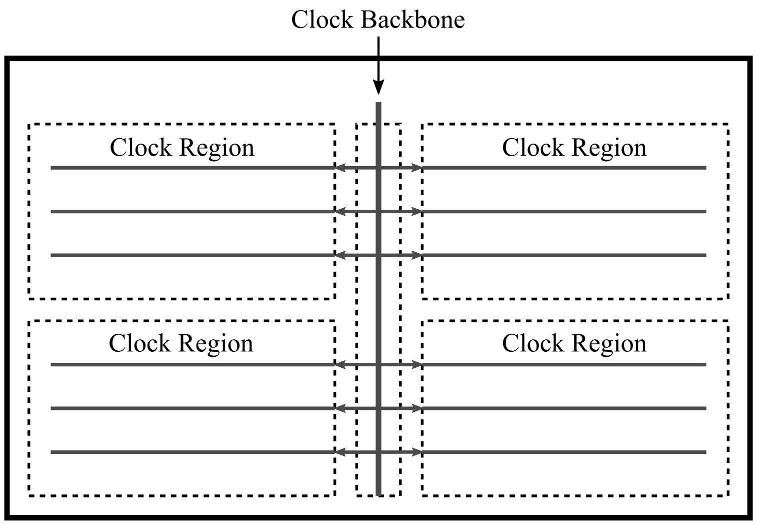

Figure 2: FPGA Clock Architecture.

with development tools to place a submodule of a soft-core processor in a desired location. More specifically, a user can choose which clock region contains a specific submodule. The placement of these submodules affects how the FPGA provides clock lines to all places in the design.

2.4 Faulting

Physical attacks target the hardware implementation in order to acquire information, alter program flow, or corrupt data, among many other possibilities. They consist of two primary categories: side-channel attacks and fault injections [6]. This section describes methods of inducing faults, with an emphasis on electromagnetic fault injection (EMFI) and its potential countermeasures.

Circuits have limits on their operating conditions. When a circuit's environment is pushed beyond these limits, it may exhibit unexpected behavior. These unexpected behaviors are called *faults*. An attacker can generate these non-ideal environments to intentionally induce faults in a process known as *faulting*. Several common methods to induce faults exist, which are described in section 2.4.1. Three subclasses encompass the methods of faulting: algorithmic modifications, safe errors, and differential fault analysis [6]. *Algorithmic modifications* involve manipulating control flow to modify behavior or results [7]. *Safe errors* seek to understand if a fault will modify an algorithm's result [8]. Lastly, *differential fault analysis* seeks to extract secret values by analyzing corrupted outputs [9]. Using such methods has made it possible to break previously secure cryptosystems, such as an AES implementation on an FPGA [10]. Timely behavior of a fault belongs to one of two categories: transient faults or permanent faults. *Transient faults* are recoverable and only affect the behavior for a limited time, but put the device into a corrupted state. *Permanent faults* damage a device or circuit irrecoverably.

2.4.1 Common Methods

Methods of fault injection fall into two categorizations: invasive/non-invasive and passive/active. Invasive attacks involve physical modifications to a circuit or device, such as decapsulating a chip, while non-invasive attacks do not require such modifications. Active attacks require the attacker to perform an action while the circuit is in operation, while passive attacks involve modifying the setup or environment prior to circuit operation. *Clock glitching*, a common non-invasive attack vector, involves an attacker modifying the clock frequency for a short time interval. A glitch can cause the chip to read unexpected data or skip instructions, amongst many other possible outcomes. *Voltage glitching* is a non-invasive attack that involves the attacker sending a low or high voltage transient into the power supply for the circuit. *Laser-based fault injection* is an invasive attack that induces faults within small regions by using a laser beam. *Micro-probing* involves placing probes on signal lines to overwrite the signal. *Temperature-based faulting* alters the circuit's environment temperature to non-ideal conditions to modify its behavior. Lastly, *electromagnetic fault injection* enables an attacker to induce current in a circuit using the principles of electromagnetic induction. Although circuits may be susceptible to several of these methods, this research focuses on electromagnetic fault injection as its attack vector [11].

2.4.2 EMFI

Quisquater and Samyde have introduced the principle of electromagnetic fault injection (EMFI) in 2002 using an EM probe connected to a camera flash to generate a high enough voltage. The result includes induced faults in a smart card processor [12]. The underlying concept is that a rapid change in magnetic flux can induce a current in a circuit or chip. Considering a single loop of wire, Eq. 1 gives the magnetic flux through the loop ϕ_B as a function of magnetic field and surface area.

$$\phi_B = \boldsymbol{B} \times \boldsymbol{S} = \|\boldsymbol{B}\|\|\boldsymbol{S}\| \cos(\theta_\perp) \tag{1}$$

No voltage would be induced in the resulting loop unless there is a change in magnetic flux through the loop. Faraday's Law of Induction models the induced electromotive force ϵ as shown in Eq. 2.

$$\epsilon = -N\frac{\delta(\phi_B)}{\delta t} = -N\frac{\delta(\boldsymbol{B} \times \boldsymbol{S})}{\delta t} \tag{2}$$

According to Eq. 2, the induced electromotive force is a product of the number of turns in the coil N and the change in flux over time $\frac{\delta(\phi_B)}{\delta t}$. In the context of EM fault injection, generating this rapid change in flux in proximity of a circuit or chip will corrupt signals and logic values. This method often does not require any modifications to the circuit; thus it is non-invasive by nature. EM fault injection is an active attack since an attacker must choose which time interval during the circuit's operation in which to generate the change in magnetic flux. Since the attacker does not need to modify the behavior of components of the circuit, such as clock signal or voltage input, EMFI is significantly more difficult to detect at runtime, and leaves no evidence of tampering [12].

2.4.3 EMFI Countermeasures

With the recognition of EM fault injection as a security threat, countermeasures have emerged in recent research. The countermeasure introduced in Miura *et al.* [13] has been capable of detecting the presence of an EM probe at distances greater than 0.1mm. The countermeasure detects frequency shifts in an LC-circuit using dual sensor coils within 1 microsecond. Dehbaoui *et al.* [14] has discovered that faults in an implementation of the AES encryption algorithm induce timing violations. A proposed countermeasure has used a monitoring delay with the assumption that a fault to the critical path also affects the monitoring delay. When the delay does not end on a predetermined clock edge, the fault is detected. This implementation has successfully detected the faults in just 10% of the targeted locations. Another countermeasure to fault attacks is redundancy in either software or hardware implementations. An operation can be calculated in parallel or sequentially, and the resulting values can be checked for equivalence before continuing execution. This document discusses maximizing the effectiveness of redundancy in FPGAs since it has not been fully explored in EMFI as a countermeasure.

2.5 Attacker Model

This research assumes that there exists an attacker attempting to to use EM fault injection against redundant FPGA modules. When conducting the EMFI attack, the malicious actor would have control over many EMFI parameters such as the voltage level of the EM pulse, the duration of the EM pulse, the phase of the pulse in relation to the clock signal, the probe location over the chip, and the probe type. Theoretically, the attacker would also have access to several resources:

- N or greater EM probes and accompanying hardware to perform EM fault injection, where N represents the level of redundancy implemented on the FPGA
- Physical access to the FPGA device
- Infinite time
- Ability to differentiate between valid and invalid inputs and outputs

With infinite time, the attacker can exhaustively search all possible combinations of time and location to perform EMFI for each level of redundancy. Fig. 3 represents this process. For each of these combinations, the attacker must create a pair of values that contains a valid input and its corresponding valid output. After sending the input and performing

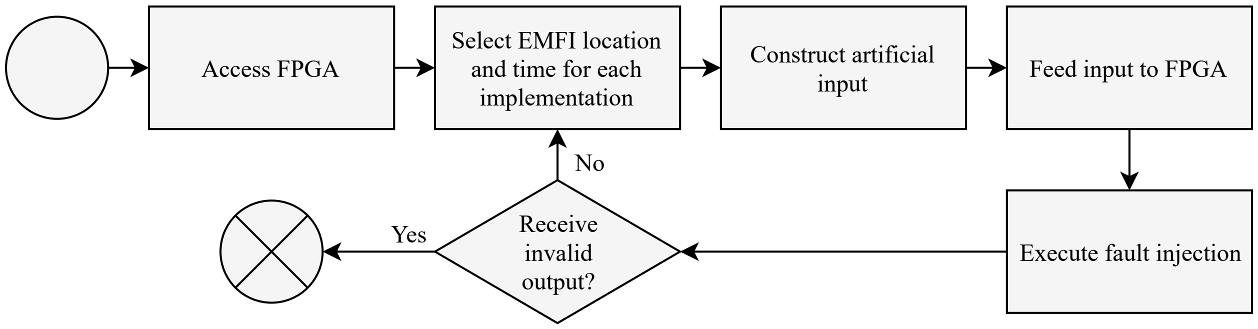

Figure 3: EMFI attacker model against redundant modules

EMFI, he or she can differentiate between valid or invalid output, an invalid output being the desired goal. There is the opportunity for valid output when less than the majority of implementations yield corrupted outputs, indicating that the device's redundancy was effective. In this scenario, there were inconsistencies between the implementations' results, allowing the device to detect the anomaly and omit the unwanted output. The process repeats until the attacker discovers invalid output, after which he or she reuses the same time and location to perform EMFI and generate additional invalid output. This research seeks to understand the ability of an attacker to reproduce identical corrupt outputs across different module layouts. If so, it will determine the magnitude of time and difficulty for a selected set of implementations.

3 Methodology

In order to investigate if altering core layout of a soft-core processor will affect fault diversification in redundancy, we sought to explore the effectiveness with both simulation and lab testing. We define the process of exploring these environments with several objectives:

1. deploy a soft-core processor
2. conduct simulations to model the behavior of various core layouts
3. analyze simulation data to select core layouts for lab testing
4. perform EMFI lab testing on the selected core layouts
5. analyze lab test data to understand the behavior of various core layouts subjected to EMFI

This section outlines the methods used to complete each objective and explains the decisions we made in regard to any hardware or software used.

3.1 Deploy a Soft-Core Processor

Throughout this research, we used of the OpenRISC mor1kx soft-core processor. The core is compliant with the OpenRISC 1000 specification and is released under the Open Hardware Description License (OHDL). The source is written in Verilog, and is made available through GitHub.

We first extracted the arithmetic logic unit (ALU) module `mor1kx_execute_alu` from the mor1kx core. This extraction enabled us to directly feed inputs and read outputs from the ALU without having to interface with other modules of the core, which were out of the context of our model. We chose the ALU as the focus of the EMFI target since the ALU is a critical module of all processors. Any program that makes use of arithmetic ultimately relies upon the ALU to execute the operation. Its behavior affects everything from the most basic of programs to complex cryptographic implementations.

The mor1kx ALU is comparable to more complex ALUs of other processors, such as the LEON3. The mor1kx ALU performs fundamental operations, such as addition and multiplication. More complex cores perform a superset of the operations that the mor1kx's ALU provides, so, we were able to model these operations without the complexities of other cores. This core allowed us to accelerate the development process using the simplicity of the mor1kx without sacrificing applicability to more complex cores. Therefore, the mor1kx is a suitable soft-core processor model in the context of this project.

We synthesized the ALU module in Xilinx Vivado 2018.1. Vivado provides the ability to specify placement of individual modules on the FPGA die. This enabled us to generate several different implementations for use in lab testing. We consistently used Vivado revision 2018.1 and mor1kx version 4.1 in development.

Vivado produced a bitstream that we deployed on a Digilent NEXYS 4 DDR development board. The board contains a Xilinx's Artix-7 `XC7A100T` FPGA. The FPGA has 101,440 logic cells, 126,800 CLB flip-flops, 4860 kilobits of block RAM, and 300 I/O pins. It was a sufficient size to hold the ALU under test. The board additionally has several peripherals, including a JTAG/UART port, a 100 MHz clock, 4 2x6 Pmod ports, 16 slide switches, 16 LEDs, and 8 7-segment displays. We were able to ensure that the ALU module is successfully deployed to the development board by performing each ALU operation with parameters derived from the switches and outputting the resulting values to the LEDs.

3.2 Conduct Simulations to Model the Behavior of Various Core Layouts

Before testing in the lab, it was important to reduce our choices for core layouts to those that are most likely to be resilient to EMFI when used together as redundant modules. Due to a large search space of an EMFI attack (phase, voltage, pulse width, probe location, probe type, etc.), it was infeasible to exhaustively test a large set of core layouts on physical hardware. Therefore, we modeled different implementations in software before we chose a set of three layouts for lab testing. To simulate the behavior of different implementations, we used a test bench in Vivado to monitor the input and output signals of the ALU.

To model various layouts, we first modified the ALU of the OpenRISC processor. The core had to be modified to accept multiple clocks to model an EMFI attack. This setup modeled the placement of submodules in multiple clock regions. In simulation, each clock region had its own clock signal. Simulation results showed an expected outcome if a fault was to affect one clock region, some combination of clock regions, or all clock regions. To accomplish modification of the core, we divided the ALU into four smaller submodules for each basic function it performed. The submodules were the adder, multiplier, divider, and a general submodule containing any other function the ALU needed to perform. When creating a simulation for a specific layout, each submodule was clocked with a signal specific to its physical clock region.

Various clock signals were created for the test bench. The first clock was correctly functioning at 10 MHz with a 50% duty cycle. This clock represents the system's normal operating frequency and was used to simulate the ALU functioning without faulting. We then created clocks to simulate a glitch caused by EMFI. These clocks composed our simulation fault search space. The search space parameters included the following:

- (A) Delay before the clock glitch
- (B) Width of the clock glitch

Fig. 4 depicts each of these parameters.

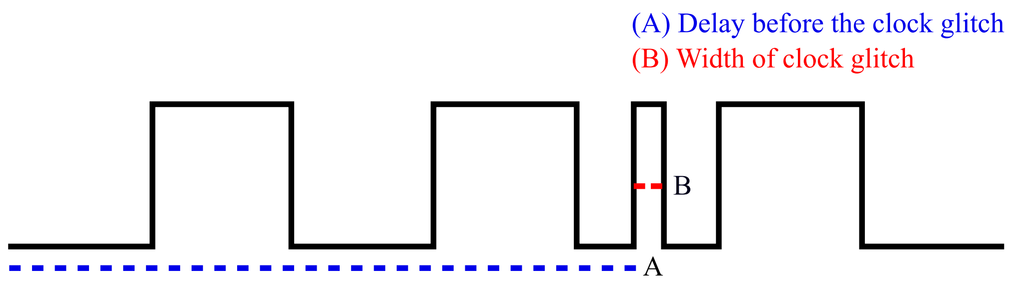

Figure 4: Clock parameters describing our search space.

Each characteristic of our faulty clock signal was varied in simulation to model different possibilities of an EMFI threat. The delay before the clock glitch (A) accomplished two things. First, delaying the glitch can change what clock cycle we were faulting. Depending on the length of time before a fault, a different ALU task would be targeted. Additionally, the delay would change the phase of the glitch within a clock period. The various phases may affect the outcome of a fault. The width of the clock glitch (B) determined the duration of a fault.

An exhaustive search of all parameters would be infeasible. When implementing simulations, we determined a subset of our fault search space. To choose this subset, we analyzed equipment characteristics that were used for lab testing and used prior knowledge about EMFI to guide our decisions. This subset needed to represent the whole fault space by ensuring the delays chosen covered all clock periods with multiple phases during ALU execution.

Next, we determined an operation to run during testing that would incorporate multiple submodules of the ALU. A simple equation, the area of a trapezoid, fits our needs because it incorporates addition, division, and multiplication. The equation for the area of a trapezoid is $A = ((b_0+b_1)*h)/2$. After deciding on this equation, we chose input values for each variable to avoid errors such as overflow. We were evaluating 32-bit operations, and therefore needed each result of the equation to be less than 2^{32}. Therefore, the following set of conditions were created for each input:

- $b_0, b_1 \leq 2^{15}$
- $h < 2^{16}$

This complete set of input conditions was infeasible to simulate. Therefore, we defined a subset of inputs that were representative of the whole input domain. The subset is described in more detail in section 4.2.

The final stage of preparation for simulations was to understand the possible core layouts of the ALU that could be simulated. These layouts encompassed a variety of possibilities, such as submodules all in the same clock region, submodules in their own individual clock regions, and combinations of different submodules located in the same clock regions. By analyzing the ALU and the submodules it is broken into, we chose a set of layouts to simulate to gain insight into which layouts would be most interesting to perform physical EMFI testing on.

After preparation, we executed our test bench simulations in Vivado. The simulation initially connected the clock signal of each clock region to the submodudules in that region. A control output set was created by providing the modules with the functionally correct clock signal. Looping through each input set in the input domain, we gathered the set of possible valid outputs produced by the current core layout. Once a control set was established, faulty clock waveforms were used.

The algorithm used to simulate all possible outcomes with the faulty clock waveforms is as follows. The outer loop iterated through the 16 layouts described in table 2. The second loop iterated through the input vectors and the innermost loop iterated through the fault search space. Once every combination was searched, the data was analyzed to determine the most successful layouts to be used in lab testing.

3.3 Analyze Simulation Data to Select Core Layouts for Lab Testing

The simulations produced records mapping input values to their corresponding output values. For each record, there was information about the time interval of the clock glitch, the targeted clock regions, and the layout of the implementation. To analyze the simulation data, we needed to standardize representations of raw data. This was accomplished by creating data processing scripts to transform the data to figures or graphics.

First, we checked the data for collisions among invalid outputs. This involved iterating through the records for occurrences of equivalent input and invalid output values across different glitch parameters or layouts. The frequency of collisions would indicate the level of fault diversification and inform selection of an implementation for testing in a lab environment.

The minimum Hamming distances was computed between invalid outputs and control outputs for every pair of layouts. We model the distribution of these distances to understand how close a trial is to producing a collision. From this information, we determined 3 layouts that are likely to show resilience to EMFI and to maximize fault diversification. These layouts would be tested in a lab environment to investigate their real-world resilience and fault diversification.

3.4 Perform EMFI Lab Testing on the Selected Core Layouts

Unlike the simulations, lab testing used an EMFI testbed to generate faults. For lab testing, we first implemented the ALU on the on the NEXYS 4 DDR development board, and assured it is functioning normally by running a test operation. This included establishing communications between the board and computer. The serial communications took place over USB using a port provided by the board. The computer would send the input value followed by a terminator. After performing the desired operation, the FPGA would send the output value over USB to the computer.

The board was placed in a Detectus HRE 41 scanner table capable of moving the EMFI probe over physical locations on the FPGA package. The scanner table provided a mount for this EMFI probe, which generated the electromagnetic field when a voltage is applied. AVTECH AVRK-4-B pulse generator produced this voltage impulse. To ensure that faults have repeatable timing, the trigger for the pulse generator was provided by the NEXYS 4 DDR board. An additional board was used to drive this signal, as the development board was not capable of doing so.

A script automated the iteration over the input and EMFI search space. While the input space is equivalent to that of the simulations, the EMFI search space comprised of a grid with 1.5 millimeter horizontal and vertical spacing over the FPGA package. On the first run, a control group was created with no fault injection. In subsequent iterations, for each input and physical location, the script positioned the probe accordingly, then sent the input over USB to the FPGA. The FPGA performed the desired operation, returning the output and triggering the pulse generator.

In contrast to simulations, we controlled the physical location of the EM probe rather than choose which submodules are affected by a glitched clock. This greatly expanded the search space as well as the possible effects that can result from EMFI. Additionally, the lab testing would encompass effects on programmable interconnects, clock trees within submodules, and FPGA-specific parameters that are not modeled in simulations.

3.5 Analyze Lab Test Data to Understand the Behavior of Various Core Layouts Subjected to EMFI

Similar to the simulation data described in section 3.3, the lab testing produced a set of input and output values, as well as associated parameters. It contained the following values:

- Input value: integer
- Output value: integer
- Fault delay: float
- Crashed: boolean
- X-coordinate: float
- Y-coordinate: float
- Layout ID: integer

First, we searched for collisions and computed minimal Hamming distances using the method described in section 3.3. In addition to these processing steps, we would create a heatmap of the minimal Hamming distances between invalid outputs for each layout to indicate if there was a trend. These distances were not limited to coordinates or fault delay across layouts, and instead, were computed amongst every invalid output pairs. We generated additional heatmaps for each layout indicating the fraction of trials resulting in a crash and an invalid output.

4 Simulating the Effects of EMFI on Various Core Layouts

The project's first objective was to simulate the behavior of different implementations of the mor1kx ALU under an EMFI threat. This process is defined in the following procedure:

1. modify the mor1kx ALU
2. define a search space
3. design a test fixture to iterate through the search space
4. analyze simulation data

This section describes the technical work completed to accomplish each task and explains how the data collected informed the decision about which implementations were tested in a lab setting.

4.1 Modifying the ALU

The ALU of the mor1kx soft-core processor was packaged as a single module called `mor1kx_execute_alu`. To allow for placement of specific groups of the ALU to create different core layouts, the first task was to divide the ALU into four modules. The module `mor1kx_execute_alu` remained as a top module with three submodules that each control an operation required by the ALU: addition, multiplication, and division. All combinational and sequential logic related to each operation was moved to the respective submodule. This left logic controlling flags and communication between each operation to the outer module.

Because faults in real world scenarios could possibly affect just parts of a design, the simulations needed to reflect this behavior. To accomplish this, the ALU needed to accept multiple clock signals as inputs. This capability allowed a specific submodule to receive a faulty clock signal while the other modules operate normally. Originally, the outer module had one clock input that was also passed along to each submodule. Once modified, the outer module contained four clock inputs. Three were dedicated to a respective submodule and the fourth was for the outer module itself.

Lastly, added functionality was given to the ALU's outer module to control the execution of our equation, $A = ((b_0 + b_1) * h)/2$. To complete the operation, a state machine with four states was developed. The first state was a waiting stage. The responsibility of this state was to record a result from the previous calculation and then pause until a trigger signal was received to start another calculation. The next three states were each responsible for executing one operation in the equation. First, b_0 was added to b_1. Then, the sum was multiplied by h. The last stage divided that result by 2. Once complete, the ALU returned to the waiting stage. Figure 5 shows a state diagram to represent this process.

4.2 Defining a Search Space

The search space for simulating an EMFI attack on various core implementations contained many layouts, fault glitches, and input values. Because it was impractical to simulate

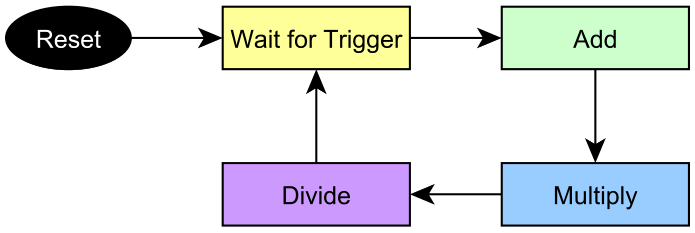

Figure 5: State diagram of ALU wrapper.

all possibilities in the search space, a subset had to be chosen that would represent the space as a whole.

With four submodules, there are 16 implementations that could be behaviorally simulated, although many more layouts exist in a physical implementation on an FPGA. The 16 variations are shown in Table 2. An *o* represents that the submodule was not faulted, and an *x* represents that the submodule was faulted. Because there were only 16 possibilities, simulations were run for each layout shown.

Table 2: Possible simulation layouts.

Submodule	Layout 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
General	o	o	o	o	o	o	o	o	x	x	x	x	x	x	x	x
Adder	o	o	o	o	x	x	x	x	o	o	o	o	x	x	x	x
Multiplier	o	o	x	x	o	o	x	x	o	o	x	x	o	o	x	x
Divider	o	x	o	x	o	x	o	x	o	x	o	x	o	x	o	x

When simulating a fault, we focused on varying the delay before the clock glitch (A) while maintaining a constant width of the clock glitch (B) at 4 ns to represent the rise time of our pulse generator. Running the ALU at 100 MHz, there was a 10 ns period in which a glitch may be present. Adjusting the delay before the glitch occurs could change which clock cycle the glitch occurs in and the phase of the glitch within the clock's period. To account for five different phases per clock cycle, the delay (A) was adjusted in increments of 2 ns for each unique simulation. Because the operation of the ALU took 390 ns to complete, the possible delays spanned from 0 ns to 390 ns. The simulations also accounted for whether the glitch caused the clock signal to be forced to a high state or a low state. At each delay, a simulation was run for each glitch possibility.

The input space chosen consisted of 10 input sets whose values satisfy the conditions in section 3.2. The sets chosen represent the whole space by accounting for edge cases as well as randomly generating some sets. Table 3 represents the ten input sets to be tested. A question mark represents a pseudorandom value that adheres to the constraints above was generated.

Table 4 shows the input sets once all the pseudorandom numbers were generated.

Table 3: Input sets chosen for simulation and testing.

Variable	Input 1	2	3	4	5	6	7-10
b_0	0	0	2^{15}	?	?	?	?
b_1	0	1	2^{15}	?	?	?	?
h	?	?	$2^{16}-1$	0	1	$2^{16}-1$	?

Table 4: Input sets with pseudorandom numbers generated.

Variable	Input 1	2	3	4	5	6	7	8	9	10
b_0	0	0	2^{15}	30920	4806	13297	1441	32620	26151	7241
b_1	0	1	2^{15}	29764	16866	24046	11120	6564	31192	12107
h	513	61287	$2^{16}-1$	0	1	$2^{16}-1$	410	35978	56072	64035

4.3 Designing a Test Fixture

To simulate various implementations of the ALU, a test fixture written in Verilog instantiated the ALU with clock signals that simulate faults occurring in different submodules. The test fixture contained four main sections, where the first one was the parameters used to instantiate the ALU. These parameters stored metadata about the current simulation and would control which clock signal each submodule of the ALU runs on.

The second section of the test fixture was a task for clock generation. When the task was called, a fork command initialized and toggled each of the clock signals at the same time intervals to create three 100 MHz signals. Two of these signals were then forced to create a glitch after a specified amount of time. One signal would force the signal to a high state for 4ns and then return to the value intended, while the second signal would force the signal to a low state for 4ns as both of these behaviors may be seen in a physical EMFI threat. Figure 6 shows the three clock signals during simulation.

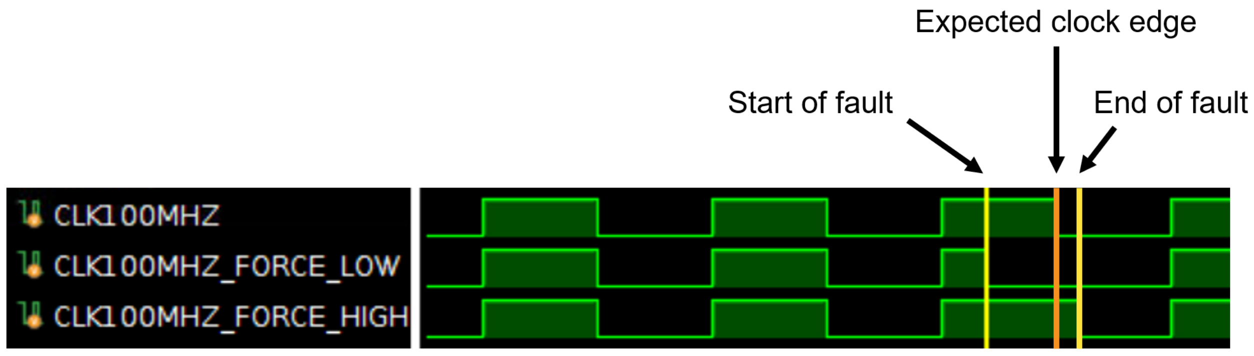

Figure 6: Three types of clock signals utilized during simulations.

Another section was responsible for controlling the execution of the test fixture and iterating through all the possible inputs. The first step of simulation execution was to load the inputs into memory. The inputs were stored as vectors in a text file and were read using the *$readmemh* command that store the vectors in an array. Then, the looping structure described in section 3.2 was implemented to iterate through the possible layouts, input sets, and faults. For each scenario, the task that generated the clocks was triggered to start the ALU's execution. When a result was returned, metadata along with the result itself was written to a comma separated values file using the *$fdisplay* command. Once recorded, a new ALU execution would be triggered for the next input set.

The final section of the test fixture added error catching by timing out if an ALU operation never completed. In some cases, a glitch in one of the clock signal could cause the simulation to run for an infinite amount of time. To allow for this behavior without having to restart the simulation, a timeout of 1 million clock cycles terminated the current operation, marked the result as a system crash, and moved to the next input set.

4.4 Analyzing Simulation Data

Simulation data came in the form of a single CSV file. While the CSV file was human readable, it lacked data types and required preprocessing before performing future computations. Thus, the data, input, and layout CSV files were cast to fixed data types, and written to several tables to a single HDF5 file. For processing, a script handled parsing and processing the data using the Pandas, NumPy, and Matplotlib packages. This allowed for all data needed for processing to be contained in the HDF5 file.

A separate method took the combined data and allocated several tables in memory to represent that data. Foreign references existed in layout and input set fields to normalize data and provide consistency across the data tables. First, the function computes the expected values for each input set. Then, for each record, determined the Hamming distance between the expected and received outputs, as well as determined if the output was corrupt. The script filtered out valid outputs, then grouped records by their output value and created an aggregate count within each group.

Figure 7 shows the number of occurrences for each corrupted output received from the simulation. All corrupted outputs were seen multiple times across the simulation search space. Eight corrupted outputs were seen four times, another eight were seen eight times, and ten were seen fifteen times. No corrupted output was seen only once.

The 100% collision rate is likely a result of running behavioral simulations rather than post-simulation or post-implementation simulations. The glitched clocks could result in the targeted modules receiving an extra clock edge. When the ALU wrapper was handling the result from a previous module, the extra clock edge could trigger the finishing module to re-execute its logic or could trigger the following module to start its logic too early. This is likely why all ten input sets had a corrupted output occur 15 times out of the possible 16 times. When the glitched clock signal was given to all modules, the clocks were still synchronized between the modules.

The results, however, would differ if the simulation were post-implementation, as the largest issue with glitched clock signals would be the propagation delays. In behavioral simulations, propagation of signals are considered instant, meaning that a glitched clock edge cannot occur during propagation. In a real world scenario, the propagation delays can be as high as the clock period (10 nanoseconds in this testing), leaving a significant window to create erroneous behavior. Additionally, unlike behavioral simulations, this leaves an opportunity for error within a module's logic. Hence, we expected to see more corrupted outputs in lab testing.

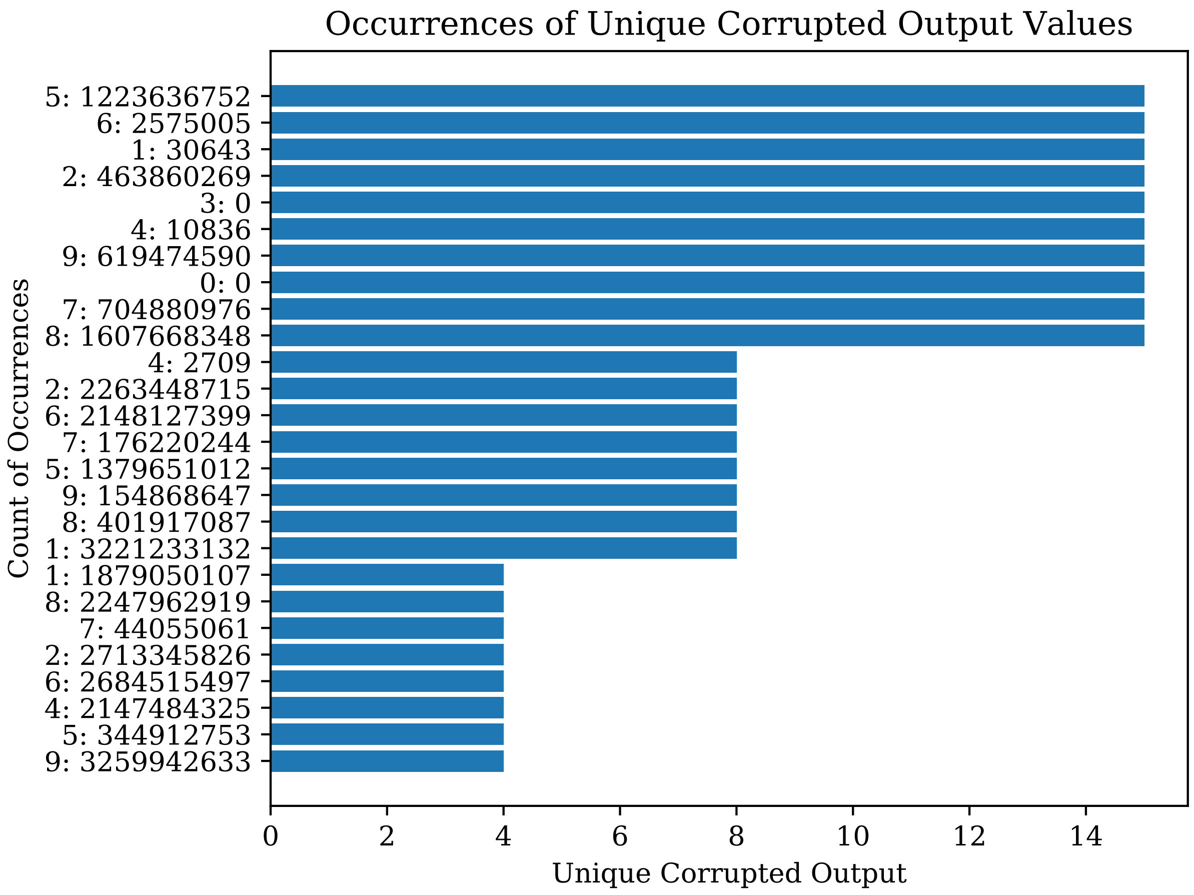

Figure 7: Occurrences of unique corrupted simulation output

5 Conducting EMFI Attacks on Physical Implementations of the mor1kx ALU

The second objective of the project is to test three physical implementations of the mor1kx ALU in an EMFI test bed. The ensuing tasks describe how this objective was accomplished:

1. develop serial communication modules
2. modify the control of the ALU
3. implement three ALU layouts in hardware
4. create a test control script
5. configure the instrumentation
6. analyze EMFI data

This section describes the technical work completed to accomplish each task and explains what results can be derived from the data collected.

5.1 Developing Serial Communication Capabilities

Communication with the FPGA board during an EMFI test was critical to data collection and test control. To accomplish this, modules were developed to create serial communication capabilities over the micro-USB connector on the board. The serial communication was split into three modules. The first, named serial interface, was responsible for receiving and sending bytes of data at a baud rate of 9600 bps with a start bit, a stop bit, and 8 data bits. The second module was responsible for controlling the transfer of more than 8 bits at a time. Specifically, this module was utilized to send a 64 bit data frame by sequentially sending one byte at a time to the serial interface module. The third module acted as a wrapper for the first two. Its task was to receive one letter commands and output flags to the ALU to execute specific tasks based on the letter received. These tasks are described more in the following section.

5.2 Modifying the ALU Control Module

In addition to developing serial communication, the control of the ALU had to be adjusted to execute when commands were received, and had to utilize the serial interface to transmit data from each test. Therefore, a wrapper module was created that instantiated both the ALU module and the serial module. It also instantiated debouncing module responsible for handling a hardware reset button on the board that sets the initial states of the modules. A hierarchy of all the modules involved is shown in figure 8.

As described earlier, the wrapper would receive flags from the serial modules when specific commands were received. The wrapper processed the flags and completed an action. When the letter r was received, all modules were reset to their initial states. When the letter t was

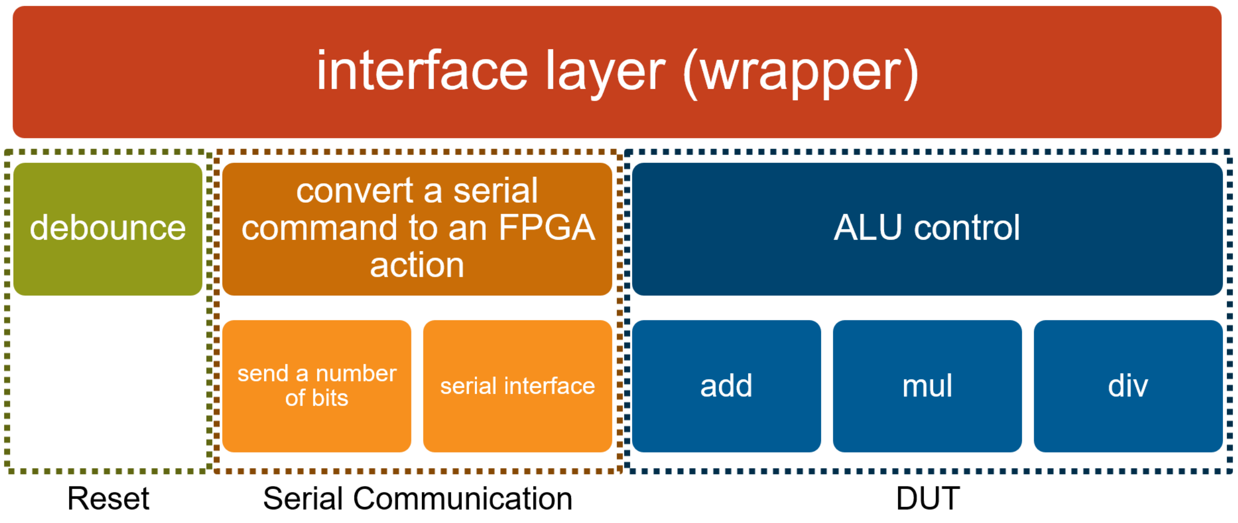

Figure 8: Verilog module hierarchy for lab testing.

received, an ALU operation was triggered causing the pulse generator to attempt to fault the FPGA during its execution. Once the operation completes, the results were sent back over serial as the 64 bit data frame. The contents of this buffer is shown in table 5. When the letter *c* was received, the same ALU operation was triggered, but instead of triggering the pulse generator a control test was run to ensure that the ALU was still functioning as intended. This needed to be done to verify that the previous fault did not cause any permanent change to the ALU modules. When the letter *i* was received, the wrapper incremented to the next input set. Utilizing these commands, all possible input sets were tested under an EMFI threat.

Table 5: Data frame used to transmit each test result using serial communication.

Bit 63-60	59	58	57	56	55-32	31-0
input set (0-9)	carry set flag	carry clear flag	overflow set flag	overflow clear flag	reserved	operation result

5.3 Implementing Various ALU Layouts

Each implementation used in physical lab testing was designed using Vivado layout tools. After synthesizing a Verilog design, individual parts of the design would be placed in a physical location on the FPGA chip. When working with these tools, you are presented with a diagram representing the FPGA chip on the target board. In this project, the Artix-7 was shown in Vivado. The diagram of the chip with each clock region labeled is shown in figure 9. When placing an element on the chip, you select the part of the design you want to place and draw a box for where Vivado should implement that part of the design. When selecting the location of various elements, it is important to consider the different clock regions available.

This project focused on three different core layouts to test under an EMFI threat. Each

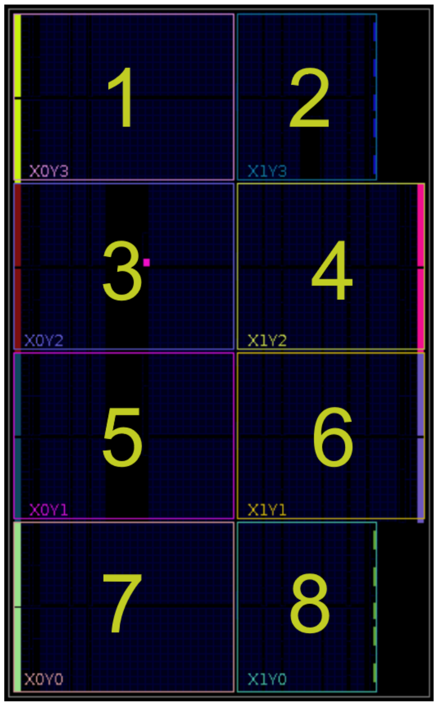

Figure 9: Clcok regions of the Artix-7 FPGA.

design placed all modules that were not intended to be faulted such as the serial communication away from the placement of ALU modules intended to be tested.

Figure 10 shows the first implementation. In this design, all ALU modules were placed together in the same clock region, and the control interfaces and serial communication modules were placed two clock regions away from that. The second implementation is displayed in figure 11, where each of the four ALU submodules were placed in a separate clock region. Once again, the control modules were placed away from the ALU. The last implementation is shown in figure 12. In this design, the ALU wrapper that controls the execution of the equation was placed in its own clock region. The three submodules responsible for each arithmetic operation were placed together in the same clock region.

5.4 Creating a Control Script

For collecting data from the hardware testbed, a script automated interaction with the scanner table, FPGA, and pulse generator. After connecting to the devices, the script iterated in order over the search space. The search space was in the outer-most loop since it required the greatest time between steps. On the other hand, updating the pulse delay on the pulse generator took only a few milliseconds. Iterating over the input spaces was the quickest, only taking a fraction of a millisecond to update. Ordering the loops with the longest steps outermost and the shortest steps innermost helped reduce the total time it took to execute the tests. From testing, the script took approximately three and a half hours to fully iterate over the search space.

For each combination in the search space, a control test was run where no fault takes

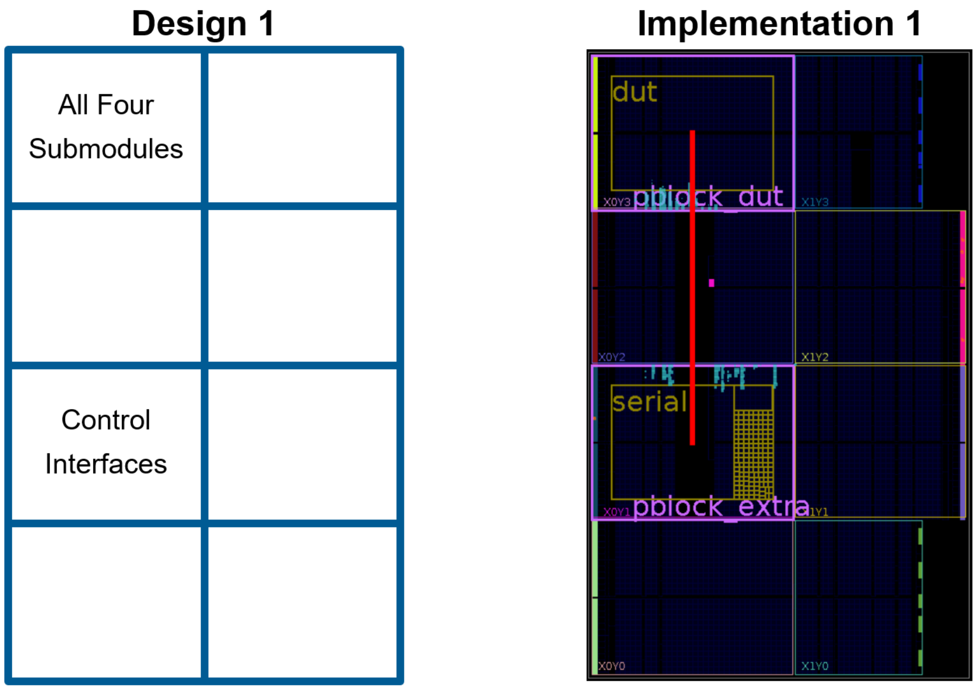

Figure 10: ALU layout 1 implemented in Vivado.

place. This is to ensure that a previous fault did not corrupt a register leading to an unpredictable state. The script would write `c` over serial, which would run the algorithm on the FPGA without triggering the pulse generator. If the script received the expected result, it would run the algorithm on the FPGA with triggering the pulse generator by sending `t` over serial. The output value was combined with the current state and added as a new row in the output CSV file.

Recovery from unexpected behavior was also a key element of the control script, as the FPGA may have an incomplete response or the pulse generator or scanner table may unexpectedly lose connection. As a solution, the script saved the current state prior to executing tests and writing test outputs. If the script crashed before writing the outputs, the user would be able to use the script with the `resume` subcommand to continue iterating over the search space. The script also offered the `skip` subcommand to assume a device crash for the current state and to skip to the next iteration. In the event that FPGA did not respond, the script would timeout and assume a device crash. Then, it would continue onto the next iteration.

5.5 Configuring the Instrumentation

The control script relied on other instrumentation to performs the tests, including scanner table, pulse generator, and probe. A crescent probe was connected in the scanner table and the ferrite was aligned with the bottom left corner of the FPGA package. We ensured that it makes gentle contact with the package surface to ensure the probe is at the minimum distance away from the die. The script would handle raising the probe prior to moving to a new physical location.

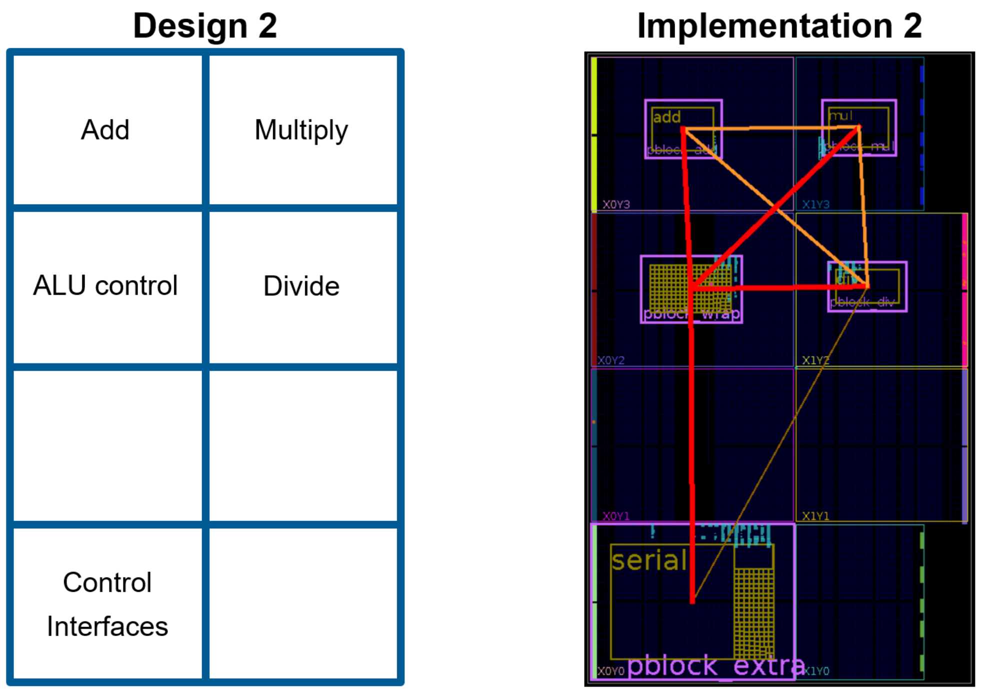

Figure 11: ALU layout 2 implemented in Vivado.

The script connected to the scanner table over the network with XML-RPC. Assuming the user had positioned the ferrite to the bottom-left corner of the package, the script set the `(0, 0)` reference point. In addition to this, the scanner table did not operate on the correct scale, so we had to apply a linear scaling before each run such that the coordinate `(15.0, 15.0)` was the upper-right corner of the FPGA package. In most runs, the scaling was approximately `(14.0/15.0, 14.0/15.0)`.

The script connected to the pulse generator over USB serial. However, before this, the user must configure the serial port on the pulse generator to operate at 115,200 baud. Then, we connected to the pulse generator using pySerial's `miniterm` to login. Once this has been configured, we proceeded with running the tests.

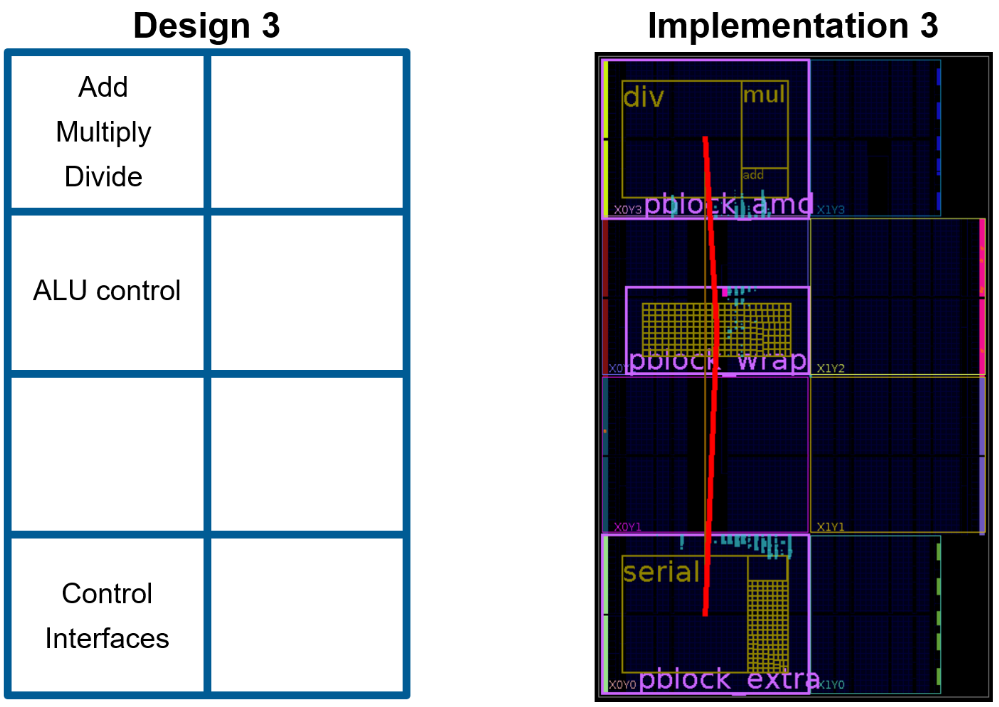

Figure 12: ALU layout 3 implemented in Vivado.

5.6 Analyzing EMFI Attack Data

Analysis and processing of the hardware testbed results begun similarly to section 4.4, where the data was parsed and aggregated. For each input set, the expected output was precomputed. Then, the Hamming distance between the received and expected outputs was computed for each record. For each layout, the data is grouped by physical location (x and y coordinates), input set, and fault delay. These groups are aggregated to provide the ratio of corrupted outputs and mean Hamming distance.

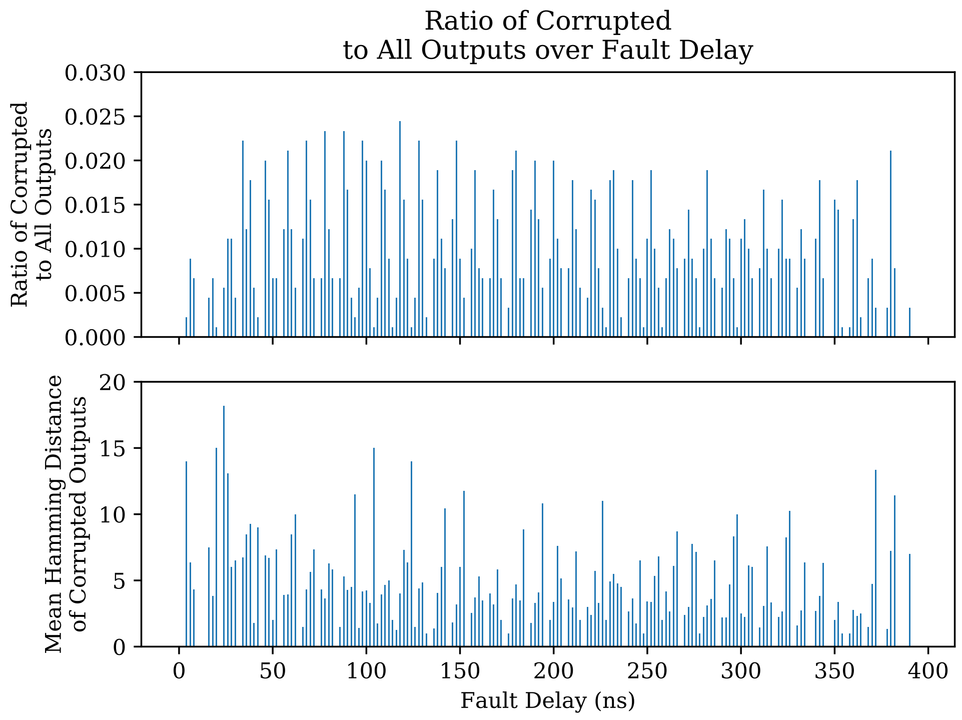

Figure 13: Fault statistics with respect to pulse delay for ALU layout 1

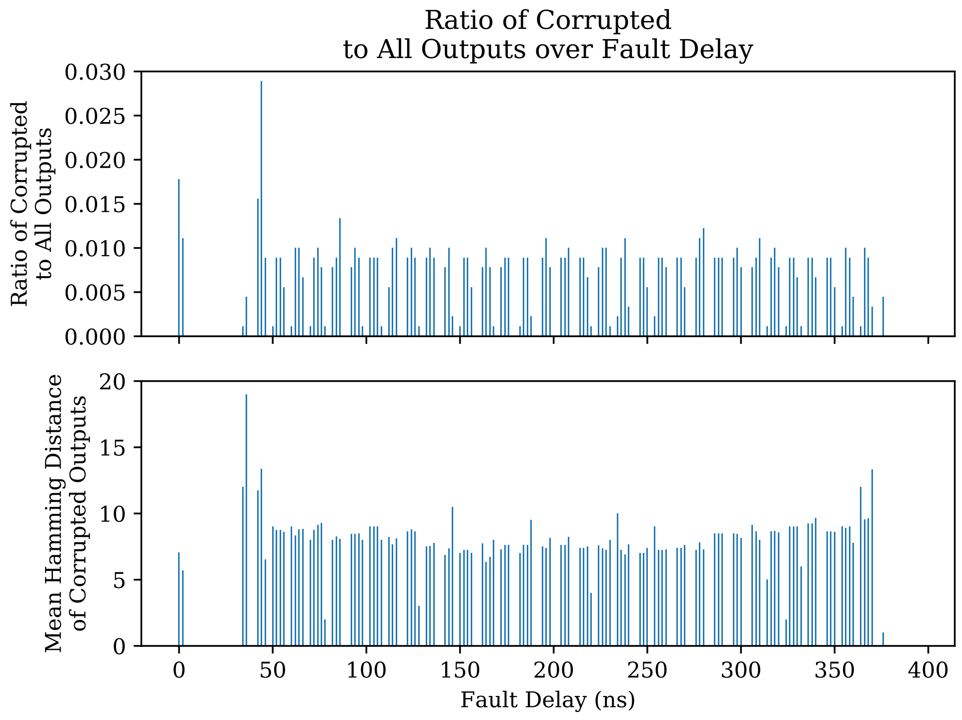

Figure 14: Fault statistics with respect to pulse delay for ALU layout 2

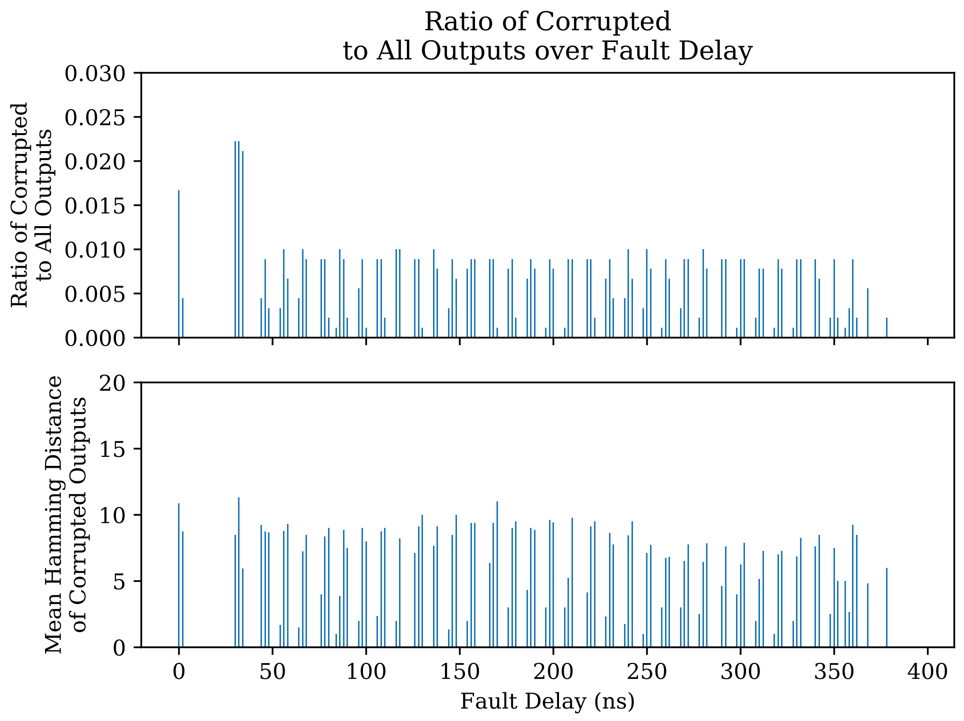

Figure 15: Fault statistics with respect to pulse delay for ALU layout 3

For the first ALU layout, figure 13 shows significantly more occurrences of faults and greater distances than for ALU layouts 2 and 3 in figures 14 and 15, respectively. Given that layouts 2 and 3 emphasize placing submodules in separate clock regions, the overall ratio of faults and distance may be proportional to utilization of separate clock regions. Also in the first layout, there is a general downward trend in the mean Hamming distances over the fault delay. This is likely a result of error accumulating through stages of the algorithm. However, this trend is not as pronounced for layouts 2 and 3.

In simulations, the prominent source of corrupted outputs was likely a result of passing data between modules. In figure 14, which shows the occurrences of faults when each module is in a separate clock region, there are considerable peaks just after 0 nanoseconds and around 40 nanoseconds. These are times when values are transferred between registers in preparation for the next module. This coincides with the trend in simulations for the occurrences in time domain. However, many other faults occurred outside of these events, which does not follow simulation results.

Additionally, when modules were in separate clock regions, ALU layouts 2 and 3 in this case, the ratio of corrupted outputs to all outputs is significantly less. While ALU layout 2 with one module per clock region shows reduced faults, ALU layout 3 with only the ALU control in a separate clock region also shows reduced faults. Comparing these layouts, the data suggests that the ALU control being in a different clock region is the most influential factor in the ratio of corrupted output occurrences.

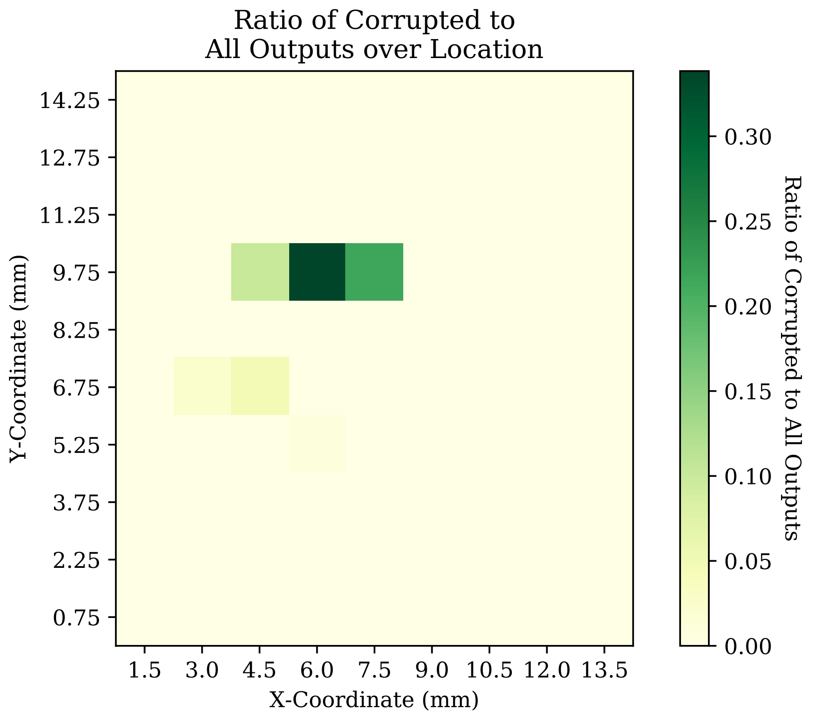

Figure 16: Fault ratios with respect to probe position relative to FPGA package for ALU layout 1

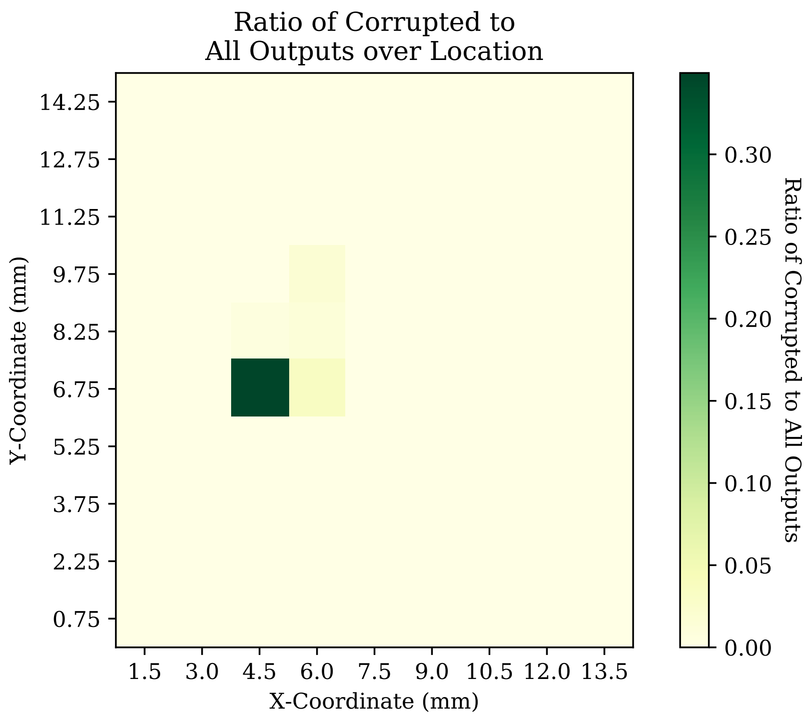

Figure 17: Fault ratios with respect to probe position relative to FPGA package for ALU layout 2

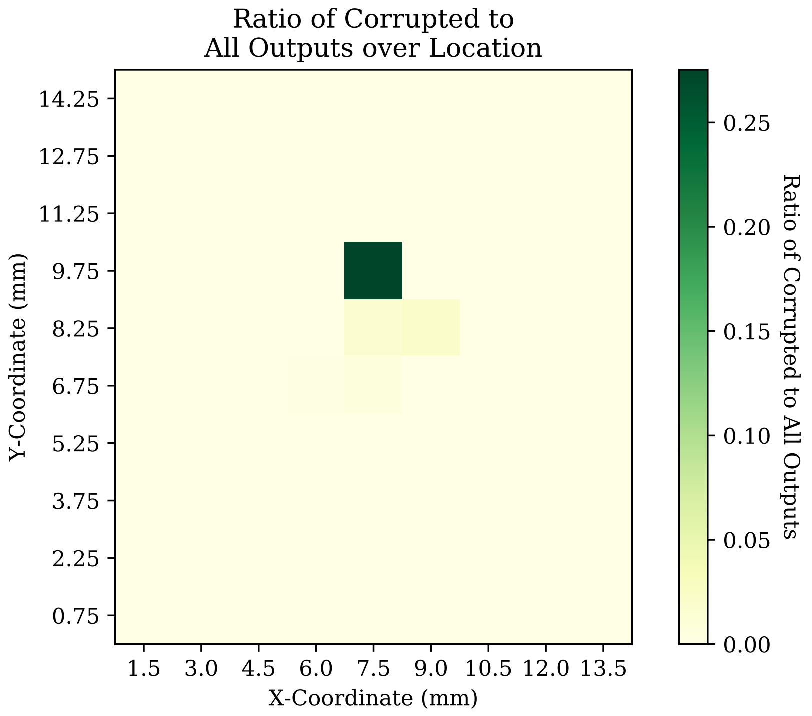

Figure 18: Fault ratios with respect to probe position relative to FPGA package for ALU layout 3

Figures 16, 17, and 18 show the ratio of faults in regards to the physical search space as a heatmap. While using the HRE scanner tables in the lab, coordinates were not consistent between runs. While we attempted to calibrate the scanner tables prior to each implementation test, the scanner table was not consistent between runs. As a result, the coordinates shown in the heat maps may not directly translate to the FPGA package. As such, we cannot draw reasonable conclusions based on these coordinates. However, the ratios shown are still accurate.

As shown in figure 16, there are two noticeable regions where faults occurred. On the other hand, in figures 17 and 18, there is only one prominent region where faults occurred. There are also regions adjacent to these where there are significantly fewer faults. While the faults in these regions may be affecting the clock lines to a module, the inaccuracies of the scanner table make it difficult to draw conclusions from the position of the regions.

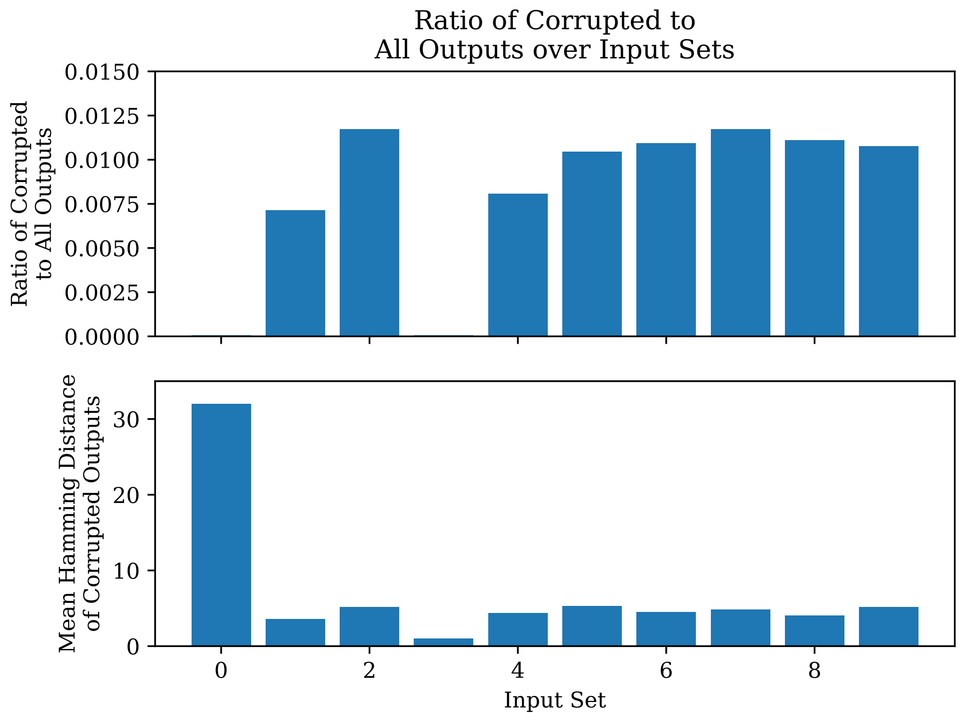

Figure 19: Fault statistics with respect to input set for ALU layout 1

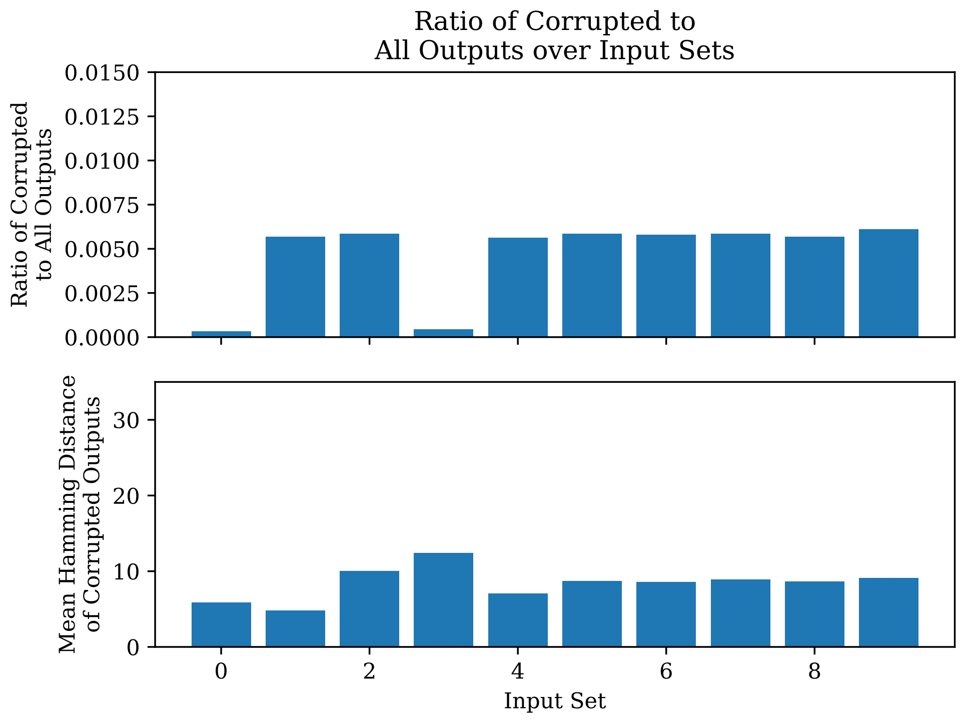

Figure 20: Fault statistics with respect to input set for ALU layout 2

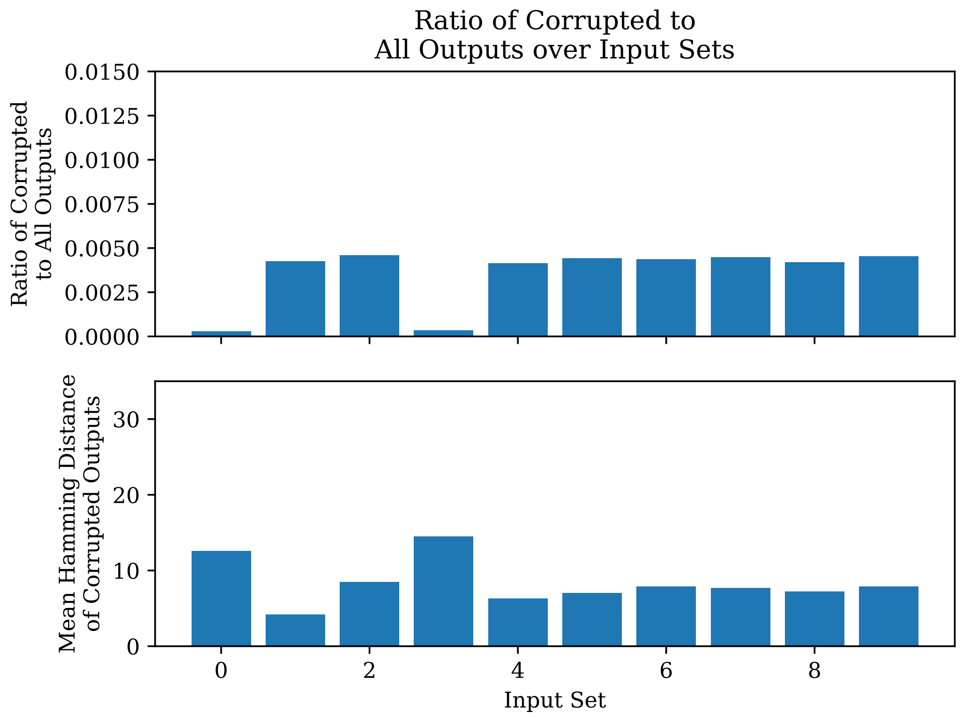

Figure 21: Fault statistics with respect to input set for ALU layout 3

In regards to the input space, input sets 0 and 3 produced a significantly lower ratio of corrupted outputs to all outputs compared to all other input sets. This is likely a result of expected result for these input sets being zero while all other results are expected to be nonzero. The ALU control module performs the algorithm in the order of add, multiply, then divide. With either both bases being 0 or the height being zero, the value being passed into the divide submodule will ideally be zero. The division submodule uses the longhand binary division algorithm; thus, when the dividend is 0, the quotient is also 0. As a result, skipping logic, such as by a clock glitch, should not produce a different result.

Amongst other potential edges cases and random input values (all inputs sets except 0 and 3), there is little difference in both ratio of corrupted outputs and mean Hamming distances. Thus, nonzero expected results have little variance among their corrupted values. Additionally, ALU layout 1 experienced a greater ratio of corrupted outputs and lower average Hamming distance compared to both ALU layouts 2 and 3. This behavior may be a result of either submodules being placed in different clock regions or longer signal lines between submodules. Yet, the lack of resolution and accuracy of the heat map make reducing this behavior to a conclusion difficult.

Table 6: Count of output types in lab testing.

Output Type	Count	Relative
Valid	531,661	99.4502%
Corrupt	2938	0.54957%
Crash	1	0.0002%
Total	534,600	100%

Table 7: Distribution of corrupt output occurrences across layouts in lab testing.

Occurrences	Count	Relative
1	1496	94.92%
2	74	4.70%
3	6	0.38%
Total	1576	100%

The number of occurrences of corrupted output values shown in figure 22 demonstrates that many collisions among corrupt outputs were discovered in lab testing. From table 6, approximately 0.5% outputs were corrupt. Of those corrupt outputs, just over 5% occurred more than once across two or more of the three layouts according to table 7. Thus, the probability of discovering a collision within our sample is 0.0549%. The probability is relatively low under the assumption that the attacker has no knowledge of implementation or layout. Given the attacker model in section 2.5, these results indicate that the assumed attacker would potentially be able to produce identical corrupt outputs for two or more different layouts. In regards to fault diversification, the statistics in table 7 show that there is a significantly greater probability of corrupt outputs that are producible in one layout, while there is a relatively low chance of identical corrupt outputs across layouts. Yet, the 5.08%

of collided corrupt outputs is a significant outcome with respect to fault similarity across layouts.

Amongst the collided outputs, there is a significant peak in collisions with respect to corrupted outputs for input set 1 shown in figure 23. For input set 4, there is a peak for mean Hamming distance with respect to all input sets. However, the mean Hamming distance for all inputs is an order of magnitude less compared to all corrupted outputs. There is also no recorded instance of a collision involving input sets 0 and 3.

Over pulse delay, there are no significant peaks of fault collided output occurrence when passing values between submodules. However, there is a peak near the end of the algorithm that most closely related to serial submodule or updating registers between the ALU control and serial modules. In terms of the mean Hamming distance, most peaks occurred within the division logic near the beginning and end. Additionally, there is a relatively consistent "floor" for mean distance outside of the peak regions.

Analyzing the peaks strictly in respect to the division submodule logic design, the most significant bits in the computation occur first, which may be a potential reason for the peak near the beginning of algorithm. Error at this point in time would, in theory, carry through the rest of the binary division, resulting in a greater Hamming distance. However, the same concept cannot be applied to the second peak region, as it occurs near the end of division.

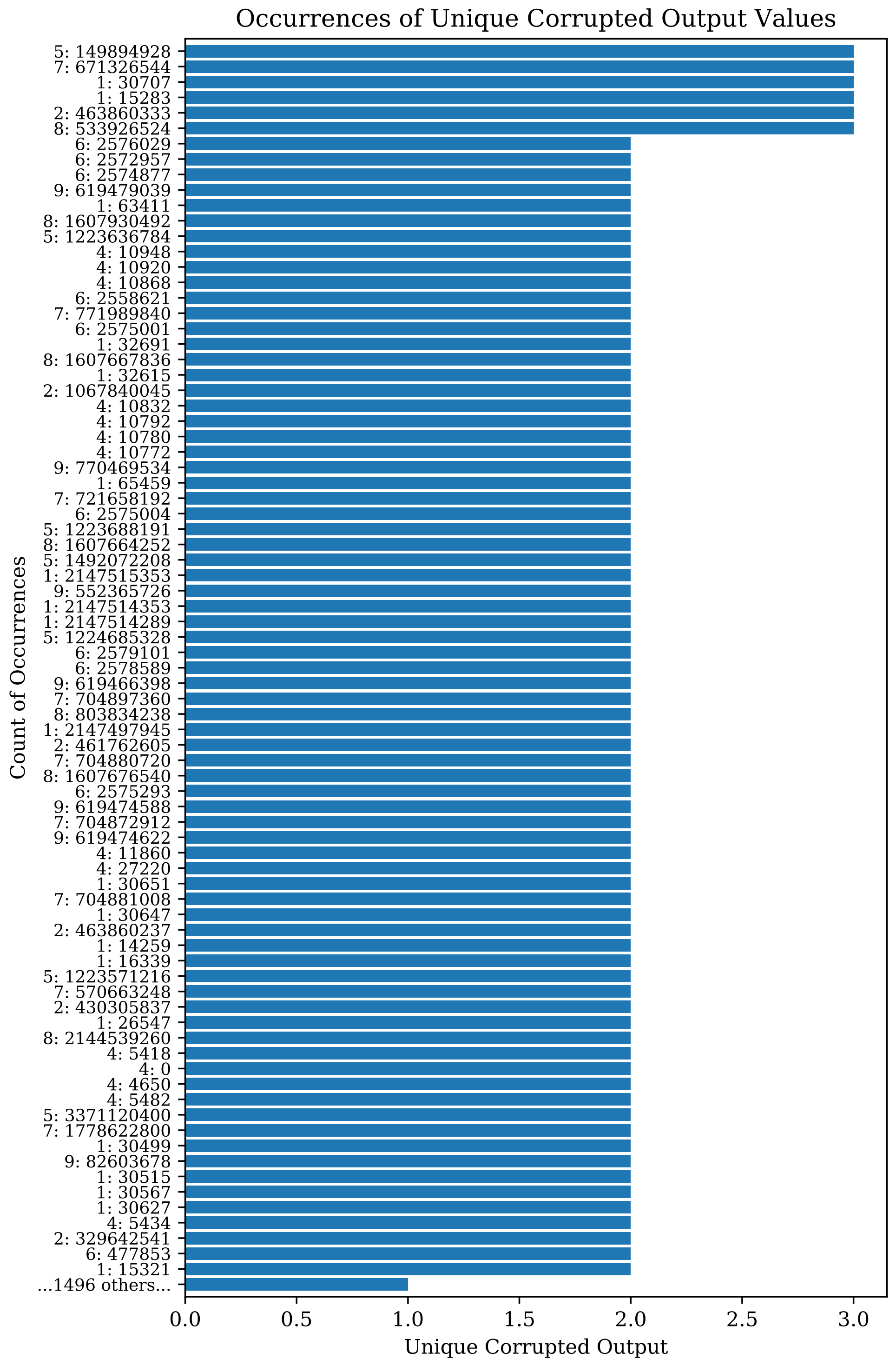

Figure 22: Occurrences of unique corrupted lab output

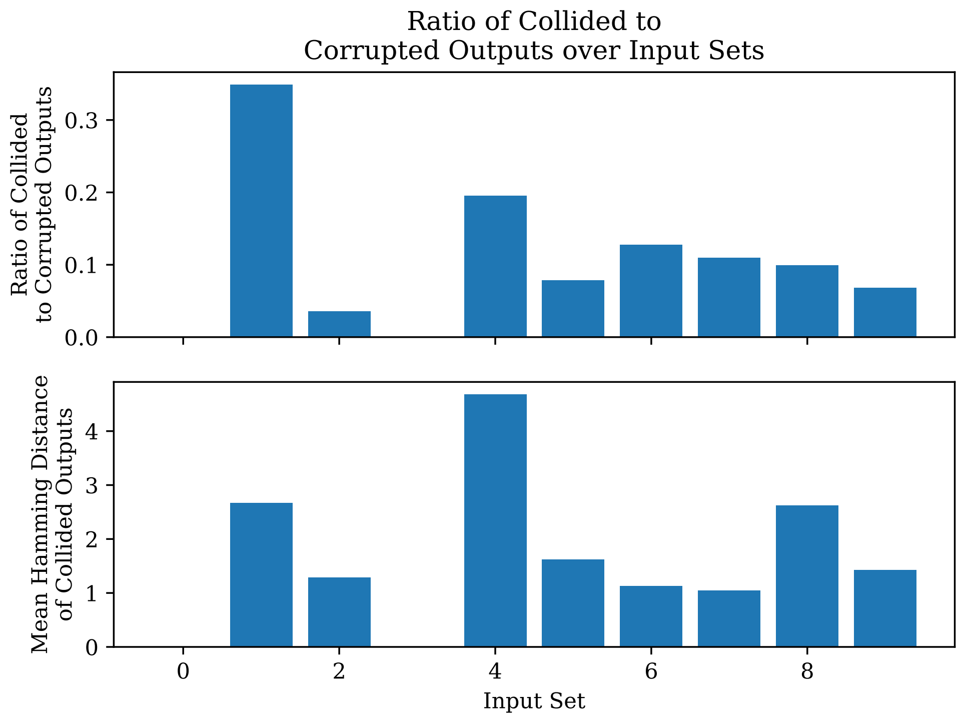

Figure 23: Statistics of collided outputs with respect to input set for all layouts

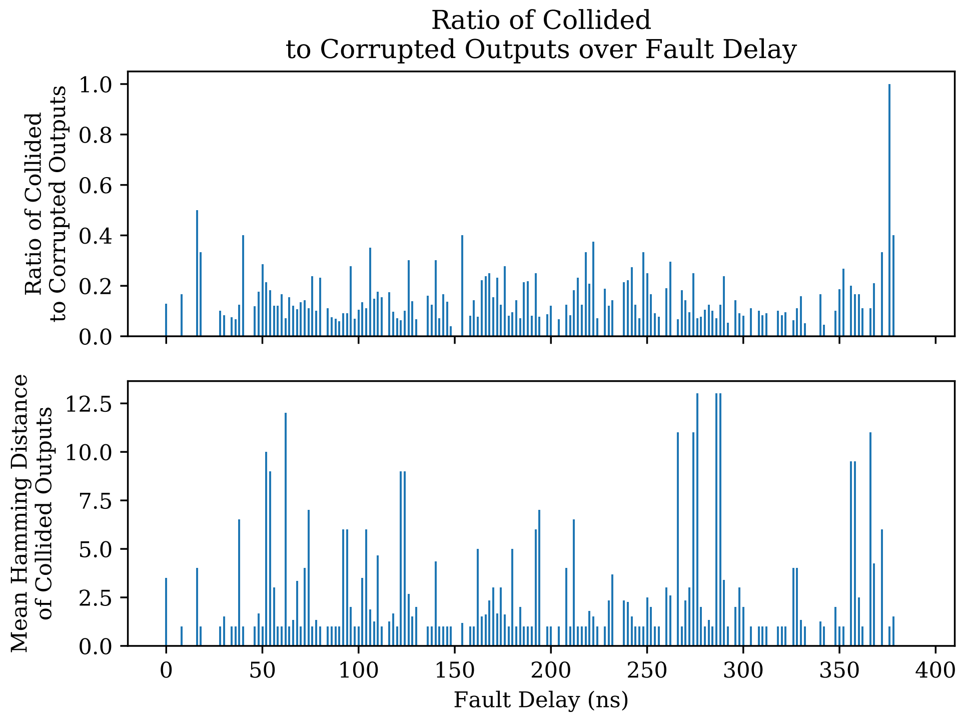

Figure 24: Statistics of collided outputs with respect to pulse delay for all layouts

6 Conclusion

In this report, we described a method for evaluating the effectiveness of fault diversification through core layout when subject to EMFI. We implemented a behavioral model for simulations to predict possible outputs in response to clock glitches. Additionally, we implemented three layouts on Xilinx's Artix-7 FPGA as a part of the Digilent NEXYS 4 DDR development board. With the three layouts, we were able perform EMFI in a lab setting within the input and physical search spaces.

The three implementations focused on separating or grouping submodules of an ALU within one or more clock regions on the FPGA. In both behavioral simulations and lab testing, the implementations exhibited corrupt behavior and outputs. However, corrupt behavior occurred at significantly more points in the input and physical search spaces. Over the entire search space, approximately 0.55% of outputs were corrupt.

We found that approximately 5.1% of corrupt outputs were identical across two or all three of the layouts. Given that the testing was designed to represent the attacker model shown in figure 3, these results are not supportive of fault diversification being considerably effective through core layout. Yet, less corrupt outputs were seen in two implementations where submodules were separated across clock regions compared to when all submodules were within the same clock region.

The results are supportive of the idea that placing submodules within separate clock regions maximizes fault diversification. Since the probability of corrupt output collisions is likely proportional to the probability of corrupt outputs, layouts demonstrating decreased ratios of corrupt outputs likely have increased fault diversification. Layouts 2 and 3, which both placed at least one submodule in a separate clock region, showed decreased ratios of corrupt outputs versus layout 1. Thus, it is plausible to maximize fault diversification by placing redundant modules in separate clock regions.

While this research makes progress towards better understanding the consequences of using redundancy schemes in FPGAs, future work can further investigate models. Beyond behavioral simulations, research into post-synthesis simulations could provide better predictions for behavior seen in lab testing, as they can take clock timing between regions and modules into account. Additionally more fine-grained research into the search space, such as the step size of the physical space or step size of the pulse delay, could better detail where and when faults are likely to occur.

References

[1] Xilinx. Field programmable gate array (fpga). [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[2] National Instruments. (2012, May) Fpga fundamentals. [Online]. Available: http://www.ni.com/white-paper/6983/en/

[3] J. G. Tong, I. D. L. Anderson, and M. A. S. Khalid, "Soft-core processors for embedded systems," in *2006 International Conference on Microelectronics*, Dec 2006, pp. 170–173.

[4] D. Arbinger and J. Erdmann. (2006, March) Designing with an embedded soft-core processor. The Plexus Technology Group. [Online]. Available: https://www.embedded.com/design/mcus-processors-and-socs/4006632/Designing-with-an-embedded-soft-core-processor

[5] Xilinx. (2018, July) 7 series fpgas clocking resources. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf

[6] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller," in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, Aug 2013, pp. 77–88.

[7] J. Schmidt and C. Herbst, "A practical fault attack on square and multiply," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, Aug 2008, pp. 53–58.

[8] S.-M. Yen and M. Joye, "Checking before output may not be enough against fault-based cryptanalysis," *IEEE Transactions on Computers*, vol. 49, no. 9, pp. 967–970, Sept 2000.

[9] A. Biham, Eliand Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology — CRYPTO '97*, B. S. Kaliski, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–525.

[10] H. Momeni, M. Masoumi, and A. Dehghan, "A practical fault induction attack against an fpga implementation of aes cryptosystem," in *World Congress on Internet Security (WorldCIS-2013)*, Dec 2013, pp. 134–138.

[11] M. Ghodrati, "Thwarting electromagnetic fault injection attack utilizing timing attack countermeasure," Ph.D. dissertation, Virginia Tech, 2018.

[12] R. Velegalati, R. Van Spyk, and J. van Woudenberg, "Electro magnetic fault injection in practice," in *International Cryptographic module conference (ICMC)*, 2013.

[13] N. Miura, D. Fujimoto, M. Nagata, N. Homma, Y. Hayashi, and T. Aoki, "Em attack sensor: Concept, circuit, and design-automation methodology," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[14] A. Dehbaoui, J. Dutertre, B. Robisson, and A. Tria, “Electromagnetic transient faults injection on a hardware and a software implementations of aes,” in *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, Sept 2012, pp. 7–15.