TU WIEN Informatics

Implementierung eines Frameworks für automatisierte Fehlerinjektion in QDI Schaltungen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Julian Spitzer, BSc
Matrikelnummer 01325893

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Florian Huemer, BSc

Wien, 1. April 2024

Julian Spitzer	Andreas Steininger

Informatics

Implementation of an Automated Fault-Injection Framework for QDI Circuits

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Julian Spitzer, BSc
Registration Number 01325893

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Florian Huemer, BSc

Vienna, 1st April, 2024

Julian Spitzer Andreas Steininger

Erklärung zur Verfassung der Arbeit

Julian Spitzer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. April 2024

Julian Spitzer

Kurzfassung

Mit immer kleiner werdenden elektronischen Komponenten, die mit niedrigeren Spannungen arbeiten, führt Strahlung mehr und mehr zu Bit-Fehlern in Komponenten und auf Leitungen. Die Robustheit gegenüber dieser unvermeidbaren Fehler ist ein wichtiger Aspekt der mit speziellen Strategien verbessert werden kann. Während es für synchrone Schaltungen viel Literatur zu diesem Thema gibt, sieht dies in der asynchronen Welt anders aus.

Eine wirkungsvolle Methode zur Quantifizierung der Robustheit ist das gezielte Einbringen von Fehlern in die Schaltung sowie die Beobachtung der Auswirkung. Dieser Vorgang wird als Fehlerinjektion bezeichnet. Diese Diplomarbeit beschäftigt sich mit existierenden Systemen für Fehlerinjektion in synchronen und asynchronen Schaltungen und geht auch der Frage nach, wann eine asynchrone Schaltung überhaupt anfällig für Fehler ist.

In weiterer Folge wird die Implementierung eines Fehlerinjektions-Systems für asynchrone Schaltungen beschrieben, welches es dem Benutzer erlaubt, Experimente an asynchronen Schaltungen durchzuführen und unterschiedliche Implementierungen zu vergleichen. Dazu wird ein Injektionsgerüst um die getestete Schaltung in Hardware implementiert, welches über eine Python-Bibliothek angesteuert wird um einen möglichst hohen Automationsgrad zu erreichen. Die besonderen Herausforderungen dabei sind das automatische Vorbereiten von Schaltungen für die Integration in das Injektions-System und die Optimierung des Systems um die Experimente mit möglichst hohem Durchsatz ausführen zu können.

Das System wird mit diversen Schaltungen (wie Addierer, Multiplizierer und einem rückgekoppelten Schieberegister) und unterschiedlichen Implementationstechniken getestet um die Brauchbarkeit zu zeigen. Zuletzt gibt die Diplomarbeit einen Ausblick auf mögliche Verbesserungen an denen in Zukunft gearbeitet werden könnte.

Abstract

With electrical components getting smaller and using lower voltages, radiation becomes an increasing factor causing unpredictable changes on wires or in components, so-called faults. The fault robustness of circuits is an important factor nowadays, and can be improved with fault mitigation techniques. While such techniques are well researched and analyzed in the synchronous domain, the same cannot be said for asynchronous circuits.

An effective method for quantifying the robustness of a circuit is the targeted injection of faults and observing the effects of these injections. This process is called fault injection. The thesis explores existing frameworks for fault injection both in the synchronous and the asynchronous design as well as taking a look at what even constitutes a fault-sensitive asynchronous circuit.

It further describes the implementation of a framework that enables the user to perform fault-injection experiments on asynchronous circuits, allowing them to draw conclusions about the robustness of different implementation techniques. This is achieved with an injection harness implemented on hardware and a Python library that works alongside to automate as many tasks as possible. The most notable challenges of this task were the integration and augmentation of given asynchronous circuits into the framework as well as optimizing the entire flow to achieve good performance when executing large experiments.

The framework is tested on various circuits (like adders, multipliers and a LFSR) with different buffer types and it is shown that it produces sensible and reproducible results. Finally, an outlook is presented, looking at potential improvements to be implemented.

Contents

CHAPTER 1

Introduction

1.1 Motivation

While synchronous design currently dominates the digital design market, the continuous reduction in transistor sizes inevitably leads to more process and timing variations, making global clock management and controlling delays ever more difficult. The inherent robustness of asynchronous circuits against such timing variations makes it seem likely that they will see a resurgence in relevance over the coming years[MN06].

Additionally, the continued shrinking of both dimensions and voltages of electronic circuits makes digital circuits significantly more sensitive to the effects of radiation [Bau05]. One of those effects are 'soft' errors where an element in the chip flips its data state, thus affecting the current execution, but not damaging the chip overall. When these soft errors occur in combinational logic, e.g., they don't directly affect a state-holding component but dissipate, we talk of Single Event Transients (SETs). SETs can turn into Single Event Upsets (SEUs) if they get captured by a state-holding element before they disappear.

Over the years, the effects of SETs have been well analyzed [ORG92][WQR$^+$04][FCMG13] with mitigation techniques being developed and tested as well [PHRB11]. A common theme of these studies is that they are generally aimed at synchronous circuits. For asynchronous circuits on the other hand it is possible to find various mitigation techniques, but their actual effectiveness is hard to compare as not a lot of studies could be found comparing the fault sensitivity in a controlled manner. This is especially troublesome, as asynchronous designs commonly use handshake signals for propagating data, making them particularly sensitive to faults.

To pave the way for such comparisons, an approach is presented in [BHNS21], where the effects of transient faults on Quasi Delay Insensitive (QDI) circuits are analyzed by large-scale simulation experiments. The sheer computational effort of running a massive amount of simulations raises the question of developing a more efficient alternative.

This thesis aims at developing a similar framework to perform large scale fault sensitivity experiments. Contrary to [BHNS21], this framework shall run the experiments in real time on actual hardware in the form of a Field Programmable Gate Array (FPGA). In addition to developing the fault-injection framework itself, the thesis will also test the framework by running a number of experiments on different circuits to compare the effect of SETs based on fault properties and implementation techniques. The execution of these experiments should be automated as much as possible to enable the analysis of millions of individual injections without the need for user interaction.

1.2 Thesis Structure

Chapter 2 gives necessary background information on asynchronous circuit types, implementation techniques and fault types to provide the base knowledge necessary for understanding the rest of the thesis. In Chapter 3 we are looking at other work that has been published that directly relates to the topics at hand.

The actual implementation details of the framework are presented in Chapter 4, covering all aspects from hardware implementation to the accompanying Python library for circuit preparation, test execution and results processing.

The circuits and additional buffer types with which the framework has been tested are shortly presented in Chapter 5 while the results of those experiment executions are shown in Chapter 6. Finally, Chapter 7 presents the accomplishments of the thesis and also lists potential improvements that could be implemented in future works.

CHAPTER 2

Background

In this chapter, some background on asynchronous circuits and their design principles are presented. This should provide a sufficient base for the further chapters of this thesis. For a deeper understanding of the matter, the reader is encouraged to consult [Spa20].

2.1 Synchronous Design

Today's digital design landscape is overwhelmingly dominated by synchronous circuits. Such designs are based on having a central clock signal which controls the chip.

In its most common form combinational logic stages are linked together by clock-controlled memory elements (usually Flip-Flops (FFs)). Based on the particular synchronous design style variant the memory elements latch the input from the previous stage on certain clock events (most commonly the rising edge). A circuit can contain multiple clock domains (e.g., different frequencies or phases) which run independently from each other but the data flow within a clock domain is entirely controlled by its associated clock.

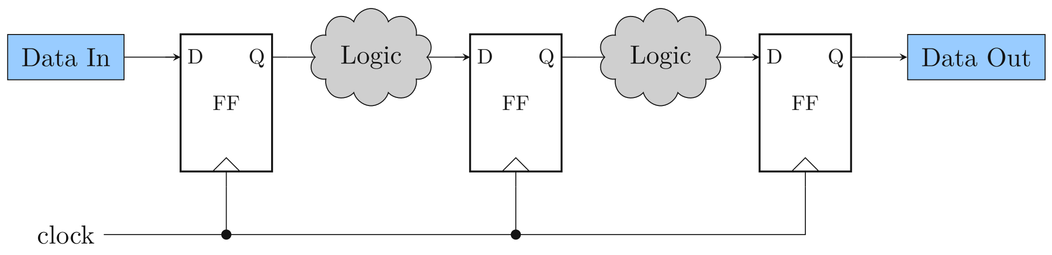


Figure 2.1: Example of a synchronous circuit.

A conceptual example of a synchronous circuit is shown in Figure 2.1. The logic clouds between the FFs are purely combinational and constitute the main constraint of a synchronous circuit: The critical path (i.e., the longest path between any two FFs) must be shorter than the clock period, thus dictating the maximum clock frequency at which the circuit can run. Furthermore, the storage elements always have setup and hold constraints, indicating the time for which the data needs to be stable before (setup) and after (hold) the active clock edge. The critical path together with the setup and hold constraints dictates the maximum clock frequency.

The biggest upside of the synchronous approach is that time can be seen as a purely discrete factor. With each clock edge all FFs in the domain switch to their next state. This abstraction simplifies circuit design and verification, as the delays of logic elements and wires do not affect the functionality and don't need to be considered for functional verification.

A drawback of synchronous design can be observed when considering the maximum clock frequency. Because the clock is global, the maximum frequency is always bounded by the longest logic path, even if all other paths are done in just a fraction of this time. Additionally, one always has to consider the worst case conditions at which the chip has to run, meaning the chip also needs to be functional at its voltage and temperature boundaries, further reducing the maximum clock frequency despite these conditions barely ever occurring.

Another challenge in synchronous designs is the complexity of managing the clock. The clock tree needs to spread across the whole chip and should reach each component at roughly the same time to be truly globally synchronous. Constantly toggling a signal on the entire chip is also a major source of power consumption. This is particularly problematic because the parts of the chip that are not currently processing data will generally still be clocked, wasting energy. A common approach for reducing the power consumption in synchronous circuits is clock gating whereby the clock is selectively disabled for currently inactive parts of the circuit [SS11][KAN11].

2.2 Asynchronous Design

As the name suggests, asynchronous designs do not rely on a global clock. Instead of latching combinational results at a fixed point in time (controlled by the clock), data is propagated in a purely event-triggered fashion. For every produced data word a handshake is performed with the consumer: The source issues a request together with valid data which the sink acknowledges when it has consumed the data[1]. This handshake mechanism is usually implemented using two wires which are referred to as request (`req`) and acknowlegement (`ack`).

This means that the individual stages do not necessarily take the same amount of time and are only directly connected to the stages right before and after. A simplified example of that is depicted in Figure 2.2.

[1]This explanation only refers to push channels. As these are the only ones used in this thesis, other channel types are not elaborated on here.

Even in the asynchronous world timing is still an important factor, swapping global clock constraints for local timing constraints between individual stages. That is the reason for the delay elements on the request lines in Figure 2.2. The delay needs to be chosen such that the associated logic cloud finishes its computation before the request signal reaches the next stage.

As the delays for each stage are chosen individually, the latency for a data word to make it through an empty pipeline is no longer bound by the global critical path between two elements. Instead, the delays between the individual stages are now bound by the local critical path connecting the two. In the case of a full pipeline the throughput is still limited by the slowest stage. Another difference to synchronous designs arises when considering voltage or temperature variations: As the entire chip can naturally speed up or slow down under differing conditions, the performance is no longer bound by worst case assumptions.

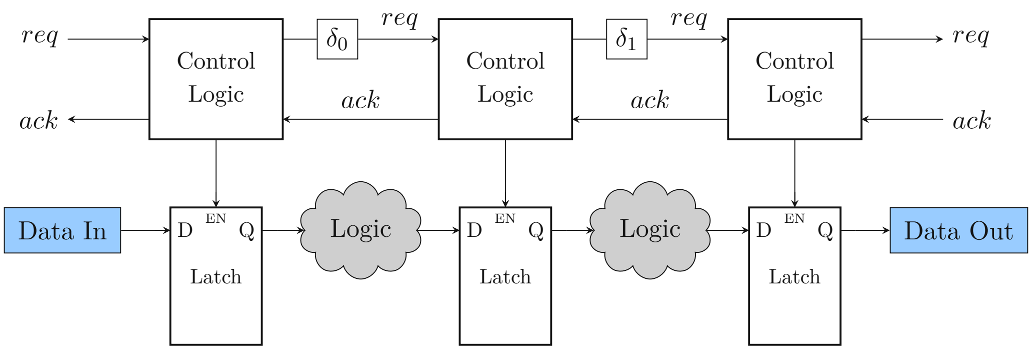
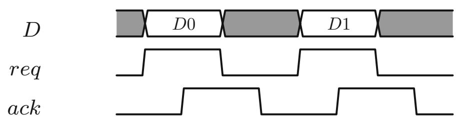


Figure 2.2: Simplified example of an asynchronous circuit.

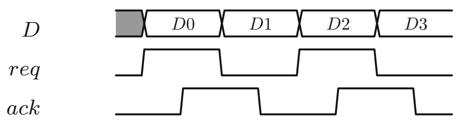
2.3 Handshaking Protocols

The `req` and `ack` signals in Figure 2.2 constitute the handshake which is always necessary in an asynchronous circuit. While the shown handshake is comprised of two explicit signals, there are also protocols where the request is implicit and only the acknowledgement requires an explicit signal.

Handshaking protocols can be implemented in two different ways: As a 2-phase or a 4-phase protocol. In 4-phase protocols a new data word is indicated by the rising edge of `req` and acknowledged with the rising edge of `ack`. This is followed by a reset phase where `req` gets deasserted followed by `ack`. This leads to the four phases of request, acknowledge, reset request and reset acknowledge. In 2-phase protocols on the other hand both edges of the handshake indicate a new data word, i.e., any edge of `req` corresponds to a new word and is followed by the same edge of `ack` to acknowledge it. The difference between the two is shown in Figure 2.3.



(a) 4-phase protocol, 4 events per word



(b) 2-phase protocol, 2 events per word

Figure 2.3: 2-phase vs. 4-phase protocol

2.3.1 Bundled Data Protocols

In Bundled Data (BD) protocols there is always an explicit request and acknowledge handshake to indicate a new data word to be consumed by the sink. The data itself can be any number of signals that is bundled together with one request signal and acknowledged all at once. This type of protocol has an inherent data race between the request and the data signals: If the request arrives before the data, the sink will consume wrong data, leading to errors. Because of this, it is always necessary to add sufficiently big delays on the request line to ensure correct functionality. The circuit shown in Figure 2.2 is an example of BD.

The biggest upside of these protocols is the low overhead as the data is transferred as-is without the need of encoding and only two additional signals are needed to make it work.

At this point the Muller C gate [Mul59] shall be introduced which is one of the most common storage elements used in asynchronous circuits. It describes an N-input logic gate with a single output. When all inputs provide the same value, it is propagated to the output. If the input signals differ, the gate holds its last output.

The C gate is particularly useful in the control path of an asynchronous circuit to store request and acknowledgement signals between stages. The so-called Muller Pipeline is built from C gates and inverters and is commonly used as the control path in asynchronous circuits, forwarding the request and acknowledge signals. An example is shown in Figure 2.4 which depicts a 4-phase BD circuit with a Muller Pipeline for the control path.

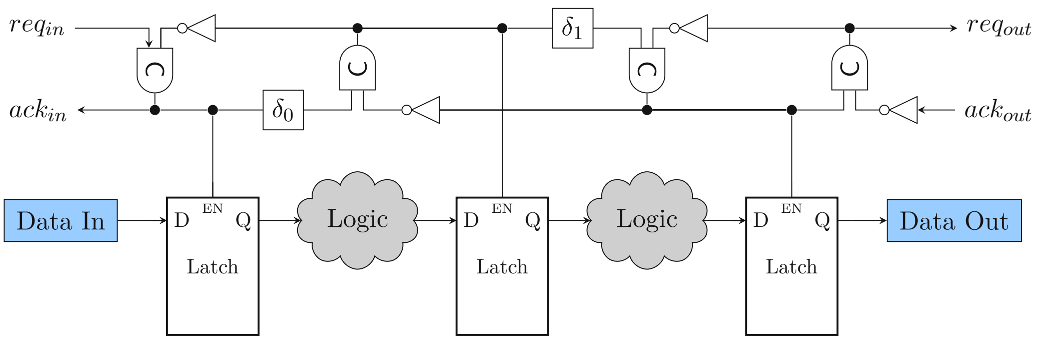


Figure 2.4: BD circuit with Muller Pipeline for the control path.

2.3.2 Delay Insensitive Protocols

Delay Insensitive (DI) protocols on the other hand do not rely on an explicit request signal but encode it into the data itself. The sink uses a Completion Detector (CD) to detect when data has arrived and can be acknowledged.

These protocols have the big advantage of eliminating the race condition between data and request. The drawback is significant overhead (both in area and performance) for the completion detection as well as for the encoding of the data.

The most commonly used encoding in DI protocols is the 4-phase **Dual-Rail (DR) encoding**. In this approach every data bit D is encoded with two rails, usually referred to as the true ($D.T$) and false ($D.F$) rails. When the data is 1, $D.T$ gets set and when the data is 0, $D.F$ gets set. When all bits of a DR channel are in a data state, the resulting data word is referred to as data token. Between two data words the protocol returns both rails to 0, providing a spacer for the completion detection to distinguish between the different phases of the protocol. In the 4-phase protocol data tokens are always separated by spacers. Having both $D.T$ and $D.F$ set at the same time is an invalid state that must never occur. Using this encoding a CD can easily check if data is valid by using an OR or NOR gate on the two rails of each bit. The implementation as 4-phase protocol is shown in Figure 2.5.

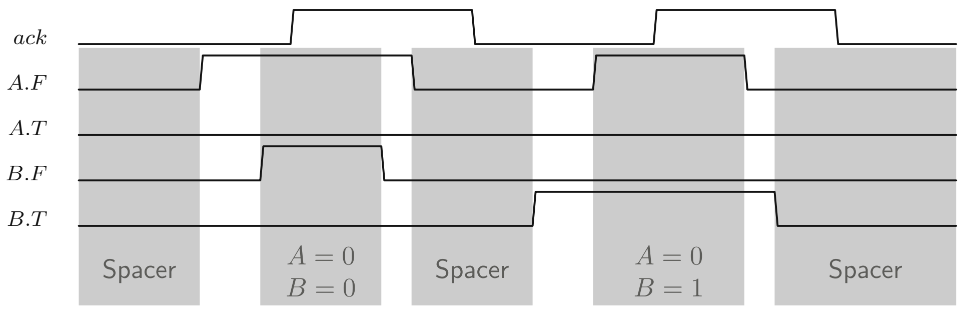


Figure 2.5: 4-phase DR example. Two bits are being transmitted in parallel.

The DR encoding can also be implemented as a 2-phase protocol. One common implementation is by not using the actual values of the rails but only observing which line has toggled. Another possibility is Level Encoded Dual Rail (LEDR) where the rails are split into a data and a parity rail, with the data rail providing the actual data and the other bit just being used for parity. The required parity (i.e., even or odd) changes between data words, so there is always exactly one rail that needs to toggle when transmitting the next word. The 2-phase protocols are generally not as popular because they cause significant additional overhead in both the CD as well as the processing of data.

2.4 Delay Modeling of Asynchronous Circuits

As time is no longer a purely discrete component, the modeling of delays is an important aspect of asynchronous design. Asynchronous circuits can be categorized into different classes based on their robustness to varying gate or wire delays.

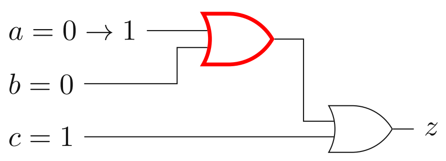
The most robust class of asynchronous circuits are DI circuits. In DI circuits, the delays of gates and wires can be completely arbitrary while still guaranteeing correct functionality. As shown in [Mar90] such circuits are limited to only using inverters and Muller C Gates.

The reason for this limitation are wire forks which is why the class of QDI circuits introduces the isochronic fork constraint. This means that the wire delays after certain forks are guaranteed to be equal. With this restriction arbitrary functionality can be implemented in a QDI way. The framework developed in this thesis is aimed at evaluating the fault robustness of QDI circuits.

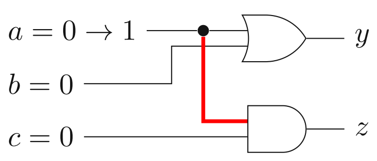
For completeness' sake also Speed Independent (SI) and Self-Timed (ST) circuits shall be mentioned. In SI circuits, gate delays can be arbitrary but all wire delays are assumed to be zero. If all forks are isochronic, SI and QDI are the same timing model. ST circuits describe the concept of circuits which require additional higher-level timing constraints to work. The data race in BD protocols constitutes such a constraint.

For a circuit to be SI, DI or QDI, it must be free from **gate and wire orphans**. A gate becomes an orphan when it toggles for some particular input but does not actually impact the observable output as shown in Figure 2.6a. When dealing with arbitrary gate delays like in the aforementioned timing models, gate orphans are problematic because a gate might toggle long after the previous phase has passed, resulting in invalid or wrong output. If gate orphans occur in combinational DR logic, specific internal completion detectors are required, whose outputs are combined to a 'done' signal that is used together with the output to verify that the circuit has finished processing and data is ready to be acknowledged.

Wire orphans can appear after wire forks when only a single destination is able to observe the signal. This is pictured in Figure 2.6b. Again, this can cause issues for arbitrary delays as the toggling wire might cause the output to toggle in an unrelated phase. The problem of wire orphans disappears with the isochronic fork constraint.



(a) Gate orphan



(b) Wire orphan

Figure 2.6: Gate and wire orphans.

2.5 QDI Circuit Design

As this thesis is mostly centered around fault injection in DR QDI circuits, one popular DR buffer and three QDI implementation techniques for combinational logic are presented in detail. The robustness of these implementation techniques will later be compared when running experiments with the framework introduced in this thesis.

2.5.1 Weak-Conditioned Half Buffer

One of the most basic ways of storing data in a QDI pipeline is with the Weak-Conditioned Half Buffer (WCHB). In this circuit C gates are used for capturing the values of the data rails. A completion detector recognizes when the C gates have switched to a valid phase (data or spacer) and sets the `ack` signal accordingly. An example is shown in Figure 2.7.

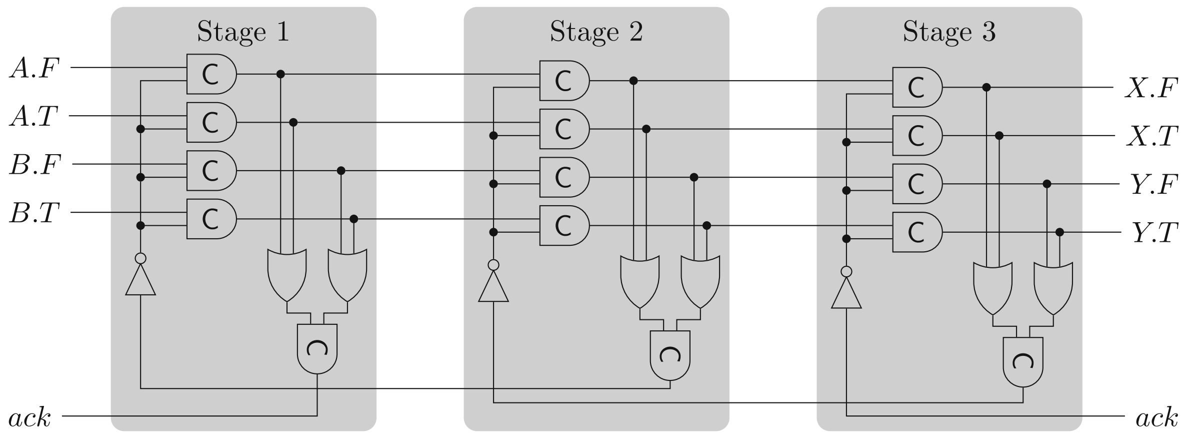


Figure 2.7: 3-stage WCHB pipeline.

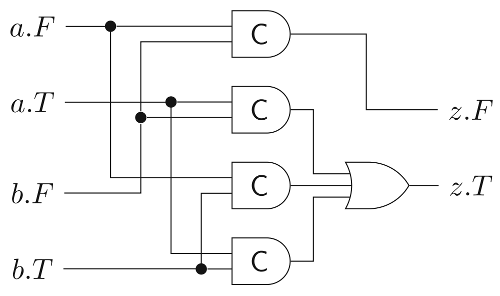
In the initial state when all data inputs and all C gates of a stage are set to 0, the inverter asserts one input of each buffer C gate. When a data word comes in, one rail of each input bit will get set, toggling the corresponding C gate. Once all bits have a rail set, the C gate of the completion detector switches to 1, indicating to the previous stage that the data has been accepted and the spacer can be sent. As the inverter is connected to the `ack` signal from the next stage, the C gates can only propagate the spacer once the next stage has consumed the data as well.

The individual pipeline stages are usually connected with combinational logic to perform data manipulation. The next sections will present design methods for such logic.

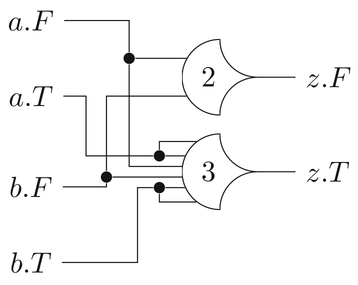
2.5.2 Delay-Insensitive Minterm Synthesis

In the Delay-Insensitive Minterm Synthesis (DIMS) [SS93] design methodology, the DR inputs are fed into a row of C gates such that there is a C gate for each possible combination of data rails (i.e., a 2-input component requires four 2-input C gates while a 3-input component requires eight 3-input C gates, etc.).

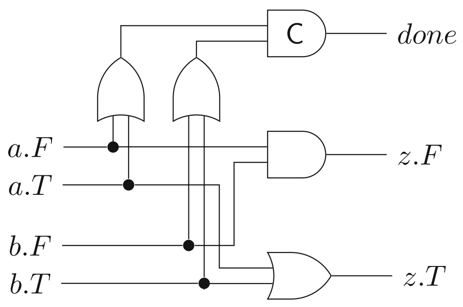
The C gates feed OR gates which directly produce the DR outputs. A DIMS implementation of an OR gate is shown in Figure 2.8a. This technique produces no gate orphans as only one of the C gates can toggle for each valid data input which in turn toggles only gates that are directly connected to the output. While it is a very robust implementation it takes up a lot of space due to the high number of C gates required.



(a) DIMS OR gate.



(b) NCL OR gate



(c) NCLX OR gate

Figure 2.8: Different implementation techniques of an OR gate.

2.5.3 Null Convention Logic

Another approach for designing QDI logic is Null Convention Logic (NCL) [FB96]. It introduces a generalized form of C gates referred to as threshold gates. These threshold gates can have any number of inputs[2] N and have a certain threshold value T. If at least T inputs are asserted, the output of the threshold gate is asserted as well. The threshold gate holds this value until all inputs are deasserted. If T is equal to N, the gate is simply a T-input C gate.

How these gates can be used to build logic functions is pictured in Figure 2.8b. It is apparent that the overall complexity of the circuit is much lower than with DIMS.

[2]In reality, there is of course an upper limit to the number of inputs due to hardware limitations.

Unfortunately, the threshold gates are expensive in area when it comes to actual hardware implementations as they require large p-stacks (N Transistors in series), making NCL not very efficient either.

2.5.4 Null Convention Logic with Explicit Completion

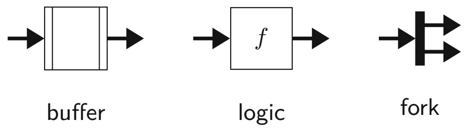
To counteract these inefficiencies, an alternative approach exists [KL02] which does not rely on threshold or C gates in the data path, making it significantly faster. In the Null Convention Logic with Explicit Completion (NCLX) approach, logic functions are implemented with non-state-holding logic gates while the input and intermediate results use OR gates for explicit completion detection. The individual completion detectors are chained together using C gates providing an explicit 'done' output to indicate that internally the circuit is done processing. The previously shown OR example in NCLX is depicted in Figure 2.8c.

The OR example is quite simple and has no internal signals requiring completion detection, but the approach scales quite well with more complex designs because every input and additional internal signal that needs completion detection requires exactly one OR gate and one C gate. While this approach still uses an inherently slow tree of C gates, they are not on the data path allowing the 'done' signal to be evaluated in parallel, improving overall performance significantly compared to conventional NCL.

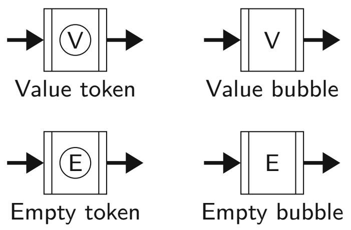
DIMS and NCLX are the two implementation techniques that will be compared in the evaluation section of this thesis.

2.6 Static Data-Flow Structures

As some concrete asynchronous circuits will be presented in this thesis, Static Data-Flow Structures (SDFSs) shall shortly be introduced. These structures allow the abstract description of asynchronous circuits, independently of the concrete implementation details. For our purposes only the 4-phase variant with a small number of operations is needed as presented in Figure 2.9a. A more comprehensive overview can be found in [Spa20].



(a) A subset of SDFS operations



(b) Tokens in SDFS

The arrows represent bundles of data that share an acknowledgement, so-called channels. Buffer blocks store valid data and spacers. Logic blocks are purely combinational and are used for data manipulation, they are usually placed between buffers in a pipeline. Forks can be used to forward a channel to multiple components at the same time, ensuring proper acknowledgement between the targets and the source.

Additionally, tokens can be represented in SDFS as shown in Figure 2.9b. Tokens are placed in buffers and represent the current state of the stored data. A value token represents a valid data word to be pushed to the next component while the empty token represents the spacer between data words. When a token is acknowledged by the next component, it turns into a bubble, meaning that the buffer is ready to accept the next input word/spacer. Analogous to the 4-phase protocol, a buffer always cycles through these 4 states.

2.7 Faults in Digital Circuits

In digital circuits a fault can be described as a signal flipping (i.e. taking on an unexpected value) temporarily on a wire (SET) or permanently (SEU) in some memory cell (e.g., a Flip-Flop or a C Gate). These faults are mostly caused by ionizing particles hitting the circuit and are unavoidable. That is why designing fault-tolerant circuits is an important aspect especially in high-radiation and safety-critical environments. The framework of this thesis is used for injecting only SETs into QDI circuits, so the injection of SEUs will not be further touched on in this chapter. However, an SET can always turn into an SEU if captured by a storage component.

In synchronous designs the effect of a transient fault is generally rather easy to predict: If the fault is short enough to have no effect at a clock edge it will simply dissipate. But if the bit flip is still present when the clock edge arrives, a wrong value will be latched and the fault propagates. Such faults can be mitigated for example with register triplication and majority voting [FCMG13].

For asynchronous circuits the effect of transient faults is a lot harder to predict. There is no global time base at which data gets propagated, every stage performs its own handshake for the most part independently of other components in the design. Additionally short pulses can cause handshakes at completely arbitrary times, potentially even leading to deadlocks. For DR designs invalid data could enter the system (i.e., both data rails of a DR signal set) or QDI properties might be violated (e.g., data rails toggling multiple times in a phase).

2.7.1 Fault Injection

Because of these difficulties it is a challenging task to properly evaluate the effects of faults on an asynchronous circuit. One of the most common strategies is fault injection at the circuit level. Generally the circuit is run on some input pattern and at some predefined time a transient fault is provoked. There are three main approaches for injecting faults: Simulation based, hardware based or emulation based fault injection [EGRM20].

Simulation based approaches are cheap to implement but require a lot of computational effort and runtime as the simulation of a circuit is very slow.

Hardware based approaches on the other hand are very fast in execution but require very expensive equipment. For hardware based fault injection the faults can be injected either through direct contact (i.e., pin-level probes) or without contact by firing external energy resources (i.e., laser beams, heavy ions, protons and neutrons) towards parts of the design [EGRM20].

The emulation based approaches provide a middle ground between the other two options, providing the speed of hardware-based approaches while still being relatively efficient. The most common approach is creating an augmented version of the circuit under test which provides some kind of interface for injecting faults and then downloading this design to a FPGA. Doing this the fault locations and injection times can be defined on software level while the actual fault-injection experiments are still run in real-time on hardware.

2.7.2 Fault Effects in DR QDI circuits

Transient faults in DR QDI circuits can lead to different kinds of errors, differing greatly in their severity. The fault-injection framework described in this thesis classifies the observations as outlined in [HNS20].

The fault might lead to a **deadlock**, where the circuit completely locks up and no longer processes any data, neither accepting input nor producing output tokens. Depending on the environment such a fail-stop behavior might be preferred over wrong data.

In the case of a **token count error**, the number of output tokens does not match the expected count. This error indicates that the fault has lead to an additional data token being generated or that an existing token has been destroyed. When the token count is lower than expected, the differentiation between deadlock and token count error can become difficult, which is why the fault-injection framework of this thesis only recognizes count errors when there are too many output tokens and only indicates a deadlock when the number of expected tokens has not been reached after a very long time.

Another possible effect are **value errors**, in which case no protocol errors can be observed at the output, but at least one output token has the wrong value. This kind of error is especially problematic as it can be very difficult to notice. This class can be compared to faults observable in synchronous designs.

A **code error** occurs when an invalid DR codeword is observed on the output, i.e., both data rails of a bit being set at the same time. **Glitches** are a similar effect where a data rail changes its value twice during a protocol phase. Both of these errors can lead to issues with completion detection or wrong values being propagated.

With **timing deviations** some output tokens appeared considerably early or late compared to a run without fault. As the insensitivity to delays is one of the main features of QDI designs, this is not an actual error.

CHAPTER 3

Related Work

In this chapter some related work on the topic of fault injection in digital circuits will be presented. In Chapter 3.1 the sensitivity of asynchronous circuits to faults is analyzed by focussing on C gates in particular. Given the heavy reliance of QDI circuits on C gates, this provides interesting insights. The observed results are subsequently compared to results obtained by our own framework. In Chapter 3.2 an FPGA based fault-injection framework generally aimed at synchronous circuits is presented. This provides good base knowledge for the principal design of such a framework in hardware. Chapter 3.3 then shows the simulation-based approach for a fault-injection framework that can be seen as the inspiration for the one developed as part of this thesis. It goes into detail on experiment execution, error classification, and techniques used for performance improvements.

3.1 On the Sensitivity of C Gates

The fault sensitivity of QDI circuits can be analyzed in an analytical way as shown by Monnet et al. [MRL04]. In their approach, the first step is to analyze when a QDI (DR) circuit is sensitive at all to transient faults. Based on this analysis a specific circuit is evaluated with different input values.

Only QDI circuits are considered that use C gates as the sole storage elements, so it is first established that a transient fault can only propagate if it causes a C gate to change its output at an unexpected time. If the fault never causes a C gate to toggle it will simply dissipate without taking effect. Because of this property it is important to exactly formalize when a C gate is sensitive to faults. For this the terminology ‘M-sensitive to 0/1’ is introduced meaning that it would take M faults on a C gate’s inputs for it to take on the wrong value.

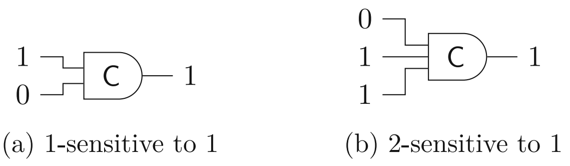


Figure 3.1: Different sensitivity states of C gates.

A 1-sensitive to 1 gate is shown in Figure 3.1a. The output would wrongly switch to 0 if there was a fault on just one input (i.e., the upper one), thus propagating the fault. It is important to point out that the circuit is not 1-sensitive to 0: If the second input would transition to 1 because of a fault, the output would remain at 1, so the fault is masked. On the other hand, the gate in Figure 3.1b is 2-sensitive to 1. There would need to be faults on two inputs at the same time to cause the C gate to switch to the wrong output, making it much less sensitive to transient faults. This leads to the conclusion that if there are currently no 1-sensitive C gates in a system, it is immune to single transient faults.

Based on this knowledge the overall sensitivity of a circuit can be analyzed. Generally speaking, a circuit's sensitivity to transient faults is related to the number of C gates that are in a 1-sensitive state and how long they remain in such states. In the paper, an in-depth analysis is provided of the fault sensitivity of a QDI 4-bit carry ripple adder implementation. The adder is tested with three different test benches, analyzing the effect of different input values:

1. Exhaustive, all input combinations were tested
2. A minimum set, with all inputs set to 0
3. A set for activating the critical path, i.e., such that the carry bit needs to ripple through all full adders.

For each test bench, the total time spent in 1-sensitive states is analyzed for all individual C gates in the circuit. The analysis shows that the sensitivity of the gates is heavily dependent on the purpose of the individual gate and also on the input data. Especially when it comes to the worst-case ripple scenario, one of the C gates spends more than 50% of the time in a 1-sensitive state making it a lot more susceptible to transient faults.

The work done in [MRL04] shows how sensitivity of asynchronous circuits can be quantified and also provides an approach for analyzing the sensitivity of particular circuits. This is a very different approach from fault injection and could be a useful tool for explaining observed behaviors in fault-injection experiments. For this thesis, the 4-bit adder circuit that has been analyzed by [MRL04] has also been recreated to run fault-injection experiments on and a comparison of results is performed in Chapter 6.3.

3.2 Fault-Injection Toolset for Synchronous Circuits

A fault-injection tool named FIDYCO (Flexible on-chip fault Injector for run-time Dependability validation with target specific COmmand language) has been developed by Rahbaran et al. [RSH04] which is aimed at synchronous designs in its basic form. In later works, this has also been extended for use with asynchronous circuits [RS08].

3.2.1 Design Considerations

The main idea for the tool is to combine hardware and software fault injection. As the target system is already available in the shape of programmable logic, it can be programmed to an FPGA, thus running in actual hardware, producing representative results in fault-injection experiments. Contrary to typical hardware-based fault injection, this can be done in an early design stage and does not require an expensive Application-Specific Integrated Circuit (ASIC) prototype to be produced. The programmable nature of the FPGA additionally allows full access to every node in the circuit, leading to high controllability and observability. As fault injections can be performed in real time, the performance is much improved over simulation based approaches.

Software-based injection frameworks generally work by injecting the faults at a code-level,requiring a recompilation and/or re-programming of the tested device. To avoid these overheads, the actual fault injector can also be moved onto the FPGA, significantly decreasing the time between individual fault injections. One downside of this approach is the area overhead from the injector, making it a difficult approach for big circuits that are already closing in on the limits of the target platform.

A typical fault-injection setup is composed of three distinct components: The fault injector module is responsible for causing the faults in the target system (Unit Under Test (UUT)), while some form of result analyzer verifies the reaction of the UUT to the injected fault. This can be done by checking against an identical target system that is running without faults ('Golden Node') or by comparing to a set of known good results. The third component is the fault-injection manager which is responsible for the selection of faults, initiating the injection and observing the results. How and where (i.e., hardware or software) to implement these components is one of the main considerations when developing a fault-injection framework based on the desired properties (e.g., performance, area efficiency, controllability).

In [RSH04], these requirements have been worked out to be met by the framework:

1. Ability to inject and observe transient faults
2. Low-level parts of the fault-injection tool to be implemented in hardware, tightly connected to UUT and Golden Node, to allow for fault injection with high precision
3. The hardware portion of the framework should be area efficient

4. Observations should be preprocessed by the hardware to reduce communication overhead

5. The user should be able to define high-level commands for the experiments, both to reduce overhead and to make the fault-injection programs more readable.

6. An automatic mode should exist which runs without user interaction

7. Experiments should run in real time

8. Failures in the value as well as the time domain should be automatically recognized by the hardware

3.2.2 The Fault-Injection Toolset FIDYCO

With consideration of the previously mentioned parameters, the FIDYCO toolset has been developed for use in FPGA platforms. This framework is a hardware/software fault-injection environment, with the hardware being implemented in Very High Speed Integrated Circuit Hardware Description Language (VHDL) and downloaded to an FPGA. The software is implemented entirely on the host to reduce chip area overhead. FIDYCO is implemented on the FPGA together with the UUT and optionally a golden node. With the use of a golden node, it is possible generate observations synchronously to the fault injections. If the user decides to prioritize area efficiency, a golden run can be used instead. In such a case, the efficiency is reduced because the fault-injection run needs to be preceded by a fault-less execution to get the golden run. The framework injects faults by parsing commands in a dynamically generated language. These commands can be be arbitrarily defined by the user and used in any order, also allowing for loops.

The framework itself can be used in interactive or automatic mode. In **automatic mode**, the sequence of commands is loaded from the ‘Local Fault Library’ (which is defined at compile time) and automatically executed. In this mode, the fault injections are performed very quickly because there is no communication overhead. In **interactive mode** on the other hand, the user can transmit a sequence of commands from the host to the FPGA which will be interpreted and stored accordingly in a specific command Random Access Memory (RAM). The actual execution then follows the same procedure as in automatic mode.

For communication to the host, a special-purpose terminal has been developed for transmitting the test sequences receiving the results of fault-injection experiments from the FPGA. The user can generally choose the interface to the host, with the framework supporting either an RS232 (Universal Asynchronous Receiver Transmitter (UART)) module or a faster Universal Serial Bus (USB) module. It is also possible to directly transfer the results to a hard disk, providing the best performance.

3.2.3 Application to Asynchronous Circuits

In a later paper, the FIDYCO concept was also used on asynchronous circuits for evaluating the robustness in comparison to their synchronous counterparts [RS08]. For that purpose, wires in the UUT were cut up and replaced with fault-injection modules, allowing for bit flips, stuck-at-0 and stuck-at-1 faults. For the purposes of their experiments, the faults were of configurable length in 50 ns steps.

Two main issues were identified when using an asynchronous UUT. On one hand, triggering the fault becomes in a 100 percent reproducible way becomes impossible as FIDYCO itself runs on a synchronous clock aligned to which the fault injections are triggered. As the UUT is inherently independent from that clock, the trigger is completely unrelated to the activities of the UUT and just based on temperature and voltage variations the fault will be applied at a slightly different point in time of the execution sequence. On the other hand, observing the effects of the injected fault becomes more difficult. The concept of a golden node is not usable because it is impossible to run two asynchronous circuits in lockstep as they are not controlled by an external signal. Because of this, the golden run concept is used instead. As the timing between two runs in an asynchronous is not reproducible, the comparisons are no longer done on a clock cycle basis but instead based on events. For bridging the gap between the asynchronous and the synchronous world, dual-clock First In First Out (FIFO) memories are used which allows for output acknowledgement without the need for synchronizers.

In conclusion, the FIDYCO framework has some similarities to the framework developed in this thesis, although generally aimed at synchronous circuits. The predictability of synchronous circuits allows for quite complex injection strategies which become more difficult to realize in an asynchronous context. That is one of the reasons why the framework of this thesis is generally kept more simple. One downside of FIDYCO is the limited error detection capability, the different error types (e.g., code errors) are not explicitly detected unless the data happens to be captured at the right moment.

3.3 Simulation Based Fault Injection on Asynchronous Digital Circuits

A large scale simulation based fault-injection framework has been developed by Behal et al. [BHNS21]. The framework is based around the idea of testing the fault robustness of a QDI circuit by injecting a single transient fault on a wire during simulation and observing its outputs. The key feature of the framework is the high degree of automation. It is designed in a way that billions of injections can be simulated in a reasonable amount of time by spreading the load over multiple nodes and storing results in a central database. The framework is used to simulate and compare the fault robustness of various asynchronous circuits implemented in different QDI design styles.

3.3.1 Experiment Steps

For the experiments a simulation model of the target circuit with annotated timing for the gates is used. This approach allows for comprehensive access to all the internal signals for fault injection and observation. The execution of a simulation experiment always follows the same outline.

Definition of Parameter Space

The first step is the definition of the parameter space to be tested. This includes obvious parameters like the circuit type or design style and spans to the input sequences to be sent to the circuit as well as the timing of the faults that should be injected. Because time in asynchronous circuits is a continuous factor it is impossible to exhaustively test all transient faults. Hence, a certain tradeoff has to be accepted when deciding on the granularity at which faults should be injected (both in start time and length of the fault).

Golden Run

To analyze the failures resulting from a fault injection it is necessary to first have a reference against which to compare. For this purpose, the circuit is first simulated on the input data without injecting any faults (i.e., golden run). This leads to a reference signal trace against which the fault-injection simulations are compared to detect deviations and, thus, errors.

Fault-Injection Run

The next step is the actual fault injection. For this the simulation is run with the same input data as the golden run but a fault is injected on one of the wires at a certain time for a specific amount of time according to the previously defined parameter space.

Failure Classification

For each injected fault the behavior of the circuit needs to be analyzed by comparing the output to the golden run. The deviations are classified as outlined in Chapter 2.7.2.

Presentation of Statistics

Finally, the results from the experiment runs need to be statistically analyzed for being able to draw conclusions about the fault robustness of QDI circuits as well as for comparing specific implementation techniques.

3.3.2 Parallelization

The generally enormous parameter space leads to a huge number of individual simulations that need to be performed. To improve performance, the execution of this workload is parallelized by the framework over multiple nodes with each node running multiple simulator instances at the same time. To reduce overhead from starting and stopping the simulator, each instance runs multiple fault-injection experiments in a row before shutting down and writing the results to the database.

For managing the simulations that still need to be performed as well as storing results, a central Structured Query Language (SQL) database is used. When starting an experiment, the database is filled with tasks (each task is a set of fault injections) to be performed according to the parameter space. Whenever an individual node is ready to perform new simulations, it independently retrieves the next task from the database and runs the associated simulations.

Every time a task has finished processing, the results of the experiments are stored in the database as well. These results not only contain the deviations but also all information that is required for re-running the simulation in case something went wrong or some interesting behavior shall be further analyzed.

3.3.3 Case Study

To evaluate the performance of the framework, large-scale experiments were executed over the course of a month, using around 40 computers in parallel. In total, more than one billion simulations were run. The tested circuits range from simple FIFOs to adders, multipliers and infinite impulse response filters (IIR). All circuits were tested with different buffer-styles, WCHB being the most basic one. Based on the experiment results, various graphs were generated, showing the frequency at which different errors occur, providing insight into the robustness of the different buffer types. Further comparisons are made, analyzing the origin of failures (i.e., either the data path or the control path), providing an in-depth insight into what parts of the circuit are susceptible to faults.

CHAPTER 4

Automated Fault-Injection Framework for QDI Circuits

This chapter introduces the emulation-based automated fault-injection framework that constitutes the main part of this thesis.

4.1 Main Objectives

The main objective for the framework is the ability to integrate a given QDI circuit (i.e., the UUT) with fault-injection inputs into synthesizable VHDL code, compile it and download it to an FPGA. It shall provide an interface to the host PC for configuring the parameter space and then running fault-injection experiments where a fixed set of input tokens are sent to the circuit while transient faults are injected. The output of the circuit shall be checked for correctness and in the case of failures, the failure is to be transmitted back to the host PC.

To reduce the complexity of the framework on one hand and because it is sufficient for our example circuits, the UUT is restricted to meet the following requirements:

- It has exactly one output channel
- It has one or zero input channels
- The input and output channels use the 4-phase DR protocol

The following high-level features shall be supported by the framework:

- Different kinds of faults (bit flips, stuck-at-0, stuck-at-1)

- Configurable durations and start times for the injected faults
- Configurable rate at which input tokens are provided
- Configurable rate at which valid output tokens are acknowledged
- Automatic augmentation with fault-injection gates of a given QDI circuit
- Synthesis and download to the FPGA
- Automated test execution based on configuration files, including results management

4.2 Framework Design

To achieve the previously mentioned goals, the framework is split into two distinctive parts:

1. Hardware Implementation (fault-injection harness in VHDL)
2. Software Interface (Python library)

At its heart the fault-injection harness is a synchronous design which acts as a wrapper around the UUT. The harness itself is implemented as a synchronous design to give full control over the individual fault-injection pulses and get a highly deterministic design which generates reproducible results. To avoid problems with metastability [Gin11] at the boundaries between the synchronous framework and the asynchronous UUT, synchronizers are used. These introduce delays on the acknowledgement of the input and the output tokens of the UUT, which are unfortunately inevitable.

The UUT around which the wrapper is built is already augmented with fault-injection gates. This is done with logic gates that are inserted at chosen wires (the victims). An example for such an augmentation is shown in Figure 4.1, depicting an OR gate where the 'A' input has been augmented with a fault-injection gate: Whenever the injection input is asserted, the XOR gate will invert the signal on the wire, causing a fault for the duration of the fault injection.

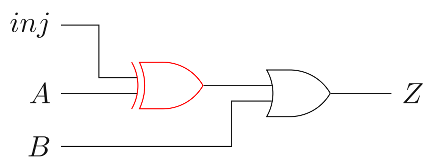


Figure 4.1: Augmentation with a fault-injection gate on wire 'A'.

For communication with the host PC, the framework provides a UART interface through which the parameters can be configured and experiments are started. For controlling this interface, a Python library has been written, allowing easy experiment execution through configuration files in JavaScript Object Notation (JSON) format. The library not only provides functions for UART access (i.e., harness configuration and experiment execution) but also for automatic augmentation of QDI circuits and the processing of experiment results.

4.3 Fault-Injection Harness

A block diagram outlining the basic structure of the fault-injection harness is shown in Figure 4.2. The block labeled 'Workstation' is the host PC which configures the framework and then launches the experiments.

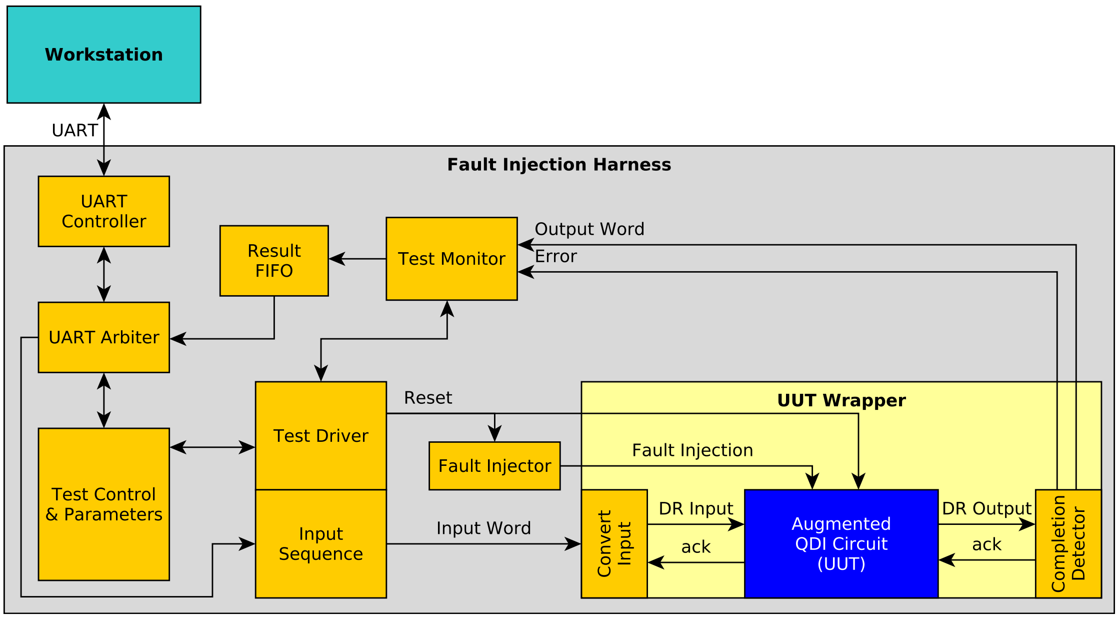


Figure 4.2: Block Diagram of the fault-injection harness. The actual QDI circuit is the block labeled UUT. Everything around it is part of the harness.

When an experiment is started, a previously stored input sequence is fed into the UUT without errors (golden run) and the output sequence is saved together with the timestamp of each output word. After that, the same input sequence is repeatedly fed into the UUT while exactly one gate is triggered to inject a fault by sending a pulse to it. The injection of the actual faults is done in a separate module running on its own clock, which is always a multiple of the core clock. This is done for all fault-injection gates and all permutations of pulse widths and pulse starting points (as defined by the test parameters). The UUT is always reset between fault injections to avoid any interference and to provide a consistent starting point for when the pulse should be injected.

During a run, the test monitor keeps track of any encountered errors. When a run has finished, the result is forwarded to the test PC via the UART interface. In general only results with errors are forwarded to the workstation to avoid the slow UART interface becoming a bottleneck. The only exception is the golden run, which is always forwarded to make it possible to receive information about the runtime of the golden run.

4.3.1 UART Interface

The **UART controller** is the direct interface to the workstation. It is a generic implementation of the UART protocol with configurable parameters. The test framework uses the following configuration by default:

- Data width: 8 bit
- Baud-Rate: 576,000
- 1 Stop bit
- Even parity

The TX/RX bytes are exchanged with the rest of the framework through simple FIFO memories, making the interface easily exchangeable if a different protocol is ever intended to be used.

Internally, the data from the UART controller goes through the **UART arbiter** which forwards the command to the right component. The component is chosen based on the first byte that is received:

- `0x00`: Test Control & Parameters
- `0x01`: Input Sequence

All further bytes are then forwarded to the selected component. That component acknowledges when it has received enough bytes for a command, after which the UART arbiter sends an acknowledgement byte back to the workstation. This kind of handshake ensures that dropped bytes would get noticed, as the host software never receives the acknowledgement.

Additionally, the UART arbiter is also connected to the result FIFO which stores the run results from the test monitor. Anytime the FIFO is not empty, the UART arbiter will read it and transmit the result to the workstation. Results are always prefixed with a byte of the value `0x80` so the host PC can easily detect it. Finally, when the last fault injection of a run has finished, it will transmit an end-of-test byte of value `0x81` to the host to indicate that the fault-injection experiment has concluded.

The full sequence of commands that need to be performed for running a fault-injection experiment is:

1. Write a sequence of input words to be fed into the UUT
2. Configure the test parameters (pulse lengths and start times on fault gates)
3. Start the experiment
4. Read the results until the end-of-test byte is received

4.3.2 Input Sequence

The Input Sequence module stores the sequence of input words that will be fed into the UUT for the actual fault-injection tests. If a circuit without input channels is tested (i.e., self-generating its output), this module is still instantiated but is simply not connected to the test circuit.

A sequence of input words can be stored by first transmitting the length of the sequence and then sending all the input words consecutively. When writing the sequence, the individual input words are always byte-aligned. The write sequence is shown in Figure 4.3.

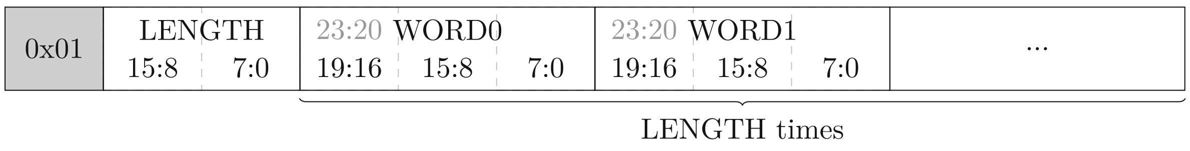


Figure 4.3: UART command structure for setting a new input sequence. The first byte is consumed by the UART arbiter and never makes it to the module. In this example, the input words are 20 bit wide which gets padded to three bytes each. Bits 23 to 20 of each word are the padding bits. When the last word has been received by the module, the UART arbiter will acknowledge it by sending 0x01 back to the workstation.

As can be seen in Figure 4.3, the length parameter is 16 bits wide so the theoretical maximum number of words for the input sequence is $65,535$. When generating the framework, the actual maximum size for the input sequence is set to a constant to reduce resource consumption on the FPGA by instantiating memories of that size.

Internally, the module is directly connected to the UUT wrapper through a read interface for feeding the input sequence into the UUT. When a run is started the module will provide the currently stored sequence sequentially until all of the words have been acknowledged or the run is aborted (e.g., in case of a deadlock).

4.3.3 Test Control & Parameters

This component allows to write and read parameters of the fault-injection framework and provides commands for starting and stopping a fault-injection experiment. The basic structure of the UART command is shown in Figure 4.4.

The supported commands are listed in Table 4.1 while the parameter addresses can be found in Table 4.2. A detailed explanation of all parameters is given in the subsequent subsections.

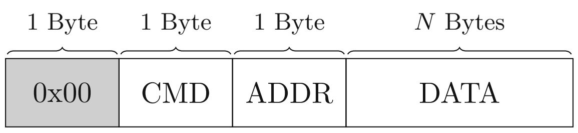


Figure 4.4: UART command structure for communicating with the test control unit. The ADDR and DATA field are optional depending on the executed command.

CMD	Command	Description
0x00	START	Start a fault-injection experiment covering the currently configured parameter space. ADDR and DATA are not used.
0x01	STOP	Stop a currently running fault-injection experiment. ADDR and DATA are not used.
0x10	READ	Read the currently configured value of a parameter. When reading, the address of the parameter is provided in the ADDR field and DATA is sent by the component to the workstation.
0x11	WRITE	Set the parameter at address ADDR to DATA.

Table 4.1: All supported commands of the test control unit. The addresses for read and write commands are listed in Table 4.2.

Fault Width

The width of the fault is one of the parameters over which the injection framework iterates and is always given in clock cycles of the injection clock. To achieve this, it is configured with three parameters: The minimum (or starting) width that is used for the first fault-injection pulse, the increment between runs, and the total number of different fault widths to be tested.

Fault Start

The second iteration parameter is the point in time where the fault starts. This too is given in clock cycles of the injection clock and provides the same three configuration parameters as the fault width: The initial starting point, the increments and the total number of starting points to iterate over. From the number of widths, starts and injection gates, the total number of injection runs in an experiment can be determined by simple multiplication.

Address	Width	Parameter
0	2	Minimum Fault Width
1	2	Fault Width Increment
2	2	Number Fault Widths
3	2	Minimum Fault Start
4	2	Fault Start Increment
5	2	Number Fault Starts
6	2	Expected Output Length
7	4	Core Clock Frequency
8	4	Injection Clock Frequency
9	1	Input Delay
10	1	Output Delay
0x80	Variable	Result RAM

Table 4.2: All parameters accessible through the test control unit. The width parameter indicates the DATA width in bytes.

Expected Output Length

By default the framework expects the same number of output tokens as input tokens are provided by the input sequence. This is the behavior when this parameter is set to 0 (which is also its reset value). When set to a non-zero value, the framework waits exactly until the configured number of output tokens was observed to consider a run finished. This is particularly useful for circuits without input channels or that produce more than one output token for each input token. When this parameter is active count errors are no longer tracked as the run immediately stops after the expected number of output tokens.

Core and Injection Clock Frequency

These parameters are fixed during synthesis and provide read-access to find the clock frequencies at which the framework runs. This is particularly useful for being able to directly convert run results into real time.

The injection clock drives only the fault injector that is responsible for generating the pulses on the injection gates. By putting this functionality in a separate clock domain, it is possible to run it at a higher frequency, allowing for higher granularity of the injection pulses.

The rest of the circuit is driven by the core clock, which is why the run durations forwarded to the workstations are given in clock cycles of the core clock.

To avoid problems with clock domain crossings both clocks need to be derived from the same source clock and the injection clock needs to be an integer multiple of the core clock.

Input Delay

This parameter allows to control the rate at which input tokens are provided to the UUT. Whenever the circuit is ready for new input (i.e., `ack` goes low), the UUT wrapper waits for the configured amount of time (in core clock cycles) before switching from the spacer to the next token. At its default value of 0, input tokens are provided as quickly as possible.

Output Delay

Analogous to the input delay, the rate at which output tokens are acknowledged can be configured. The configured delay corresponds to the number of core clock cycles between the time that a valid output word is detected and the time when it is acknowledged.

Result RAM

After a golden run execution it is possible to read out the result tokens for the stored input sequence starting with this address. To do so, the requested address of the result RAM is added to the base address of 0x80. As the address field is just one Byte and the highest bit is already used, only the results for sequences of up to 128 entries can be read out completely.

4.3.4 Test Driver

The actual execution of fault-injection experiments is managed by this module. Whenever an experiment is started (i.e., through the test control unit), the test driver performs the following steps:

1. Start the golden run by removing the UUT reset, starting the input sequence and informing the test monitor (see also Chapter 4.3.5) about the run that was started.
2. Wait for the test monitor to indicate that the run has finished.
3. From here on out the following steps are repeated until the configured parameter space has been processed:
    a) Put the UUT into reset and provide the next fault-injection parameters to the fault injector.
    b) Repeat step one (start the run), additionally start the fault injector.
    c) Wait for the test monitor to indicate that the run has finished.
4. After the last run, assert a `done` signal to the UART arbiter for it to generate the end-of-test symbol.

To avoid data loss due to FIFO overflows, the module uses a `stall` input which is connected to the result FIFO. Anytime the FIFO is almost full, `stall` gets asserted, prohibiting the test driver from starting a new run.

4.3.5 Test Monitor

The test monitor keeps track of the output tokens that are produced by the UUT. It also monitors the elapsed time between tokens and of the total run.

For each run, the test monitor generates a result containing information about the injected fault (i.e., gate location, duration and width), the run duration and any observed deviations from the golden run.

During the golden run, all output tokens received from the UUT are stored sequentially in RAM together with a timestamp. When the expected number of tokens has been received (either corresponding to the length of the input sequence or to the explicitly configured value), the result is pushed to the result FIFO.

In all further runs, the previously written output tokens are read from the RAM and compared to the tokens received from the UUT. The test monitor keeps track of deviations in the data as well as timing deviations beyond a fixed number of clock cycles. Once no output tokens have been received for a certain amount of time, the run is considered to be done. If not enough output tokens were received in that time frame, the run is marked as deadlock.

A finished run is indicated to the test driver so it can start the next iteration. In case of deviations from the golden run, the run result is also pushed to the result FIFO for transmission to the host PC.

4.3.6 UUT Wrapper

The UUT wrapper is built around the augmented QDI circuit and does the actual interfacing to the UUT.

On the input side, the next input token is continuously fetched from the input sequence and converted to DR notation. The input tokens are then provided to the UUT following the 4-phase DR protocol.

The UUT output is connected to a completion detector which generates the appropriate acknowledgement as required. Additionally, the CD keeps track of any protocol violations by detecting glitches (i.e., a data rail changes its value twice during a protocol phase) and code errors (both rails true at the same time). As the CD is running in the synchronous domain, there is the possibility of some glitches and code errors not being detected if they are shorter than the clock period. This is unfortunately not avoidable with the current implementation.

The fault-injection signals generated by the fault injector are unmodified by the wrapper and directly connected to the UUT.

4.3.7 UUT

The asynchronous QDI circuit to be tested is augmented with fault-injection gates. As previously described, these fault-injection gates take two inputs: The original wire as well as a dedicated injection wire. Based on the chosen fault type, the injection gate could cause a bit flip on the wire or a constant ‘0’ or ‘1’ for the fault duration.

While the framework supports the integration of a pre-made VHDL instance including the fault-injection gates, it also provides tool support for automatic wrapping of augmented circuits or even first augmenting a given QDI circuit with the injection gates. The different options provided by the software framework are described in detail in Chapter 4.4.

4.4 UUT Preparation

Here the components of the Python library are described which can automatically generate augmented UUT instances along with all required files.

4.4.1 PYPR

In this section the `pypr` python package[1] is introduced as it plays an integral part for automatically generating and augmenting QDI circuits through the software framework. The package has been developed by Florian Huemer to conveniently create, analyze, modify and simulate asynchronous (and specifically dual-rail QDI) circuits[Hue22].

On a lower level, digital circuits are represented in a Production Rule Set (PRS) based hardware description language. This language allows to describe components as a series of wire assignments, which can instantiate other components or logic gates directly. A simple example of a circuit description in PRS is shown in Listing 4.1.

```
prs dims_and is
    inputs a : DRBit; b : DRBit;
    outputs x : DRBit;
begin
    cg_00 := cgate(a.F, b.F);
    cg_01 := cgate(a.F, b.T);
    cg_10 := cgate(a.T, b.F);
    cg_11 := cgate(a.T, b.T);
    x.T := wire(cg_11);
    x.F := or_gate(cg_00, cg_01, cg_10);
end prs;
```

Listing 4.1: PRS example of a simple DIMS AND Gate

On a higher abstraction level, `pypr` can be used to describe circuits in Python in the form of Data Flow Graphs (DFGs). This allows the creation of QDI circuits by

[1] `https://git.ecs.tuwien.ac.at/eda/pypr`

connecting individual building blocks (given as PRS) directly in Python, simplifying the parametrization of the circuit (e.g., different buffer or logic styles, bus widths).

To further simplify the generation of QDI circuits, `pypr` has the capability of directly synthesizing combinational logic provided as Verilog functions into DR logic. Included with the library there are three logic styles with which the Verilog logic can be synthesized: DIMS, NCL and NCLX.

Another feature of the library is the possibility of exporting a given PRS circuit directly into Hardware Description Languages (like VHDL) for use in conventional Electronic Design Automation (EDA) software. This can either be done for simulation tools where the individual gates will simply be represented by, e.g., generic VHDL processes or directly for hardware in which case the logic gates are directly mapped to components of the target platform (e.g., Lookup Tables (LUTs) in an FPGA).

The possibility of importing PRS files and having direct access to all assignments is extensively used by the framework to take an existing circuit in PRS form and augmenting it with fault-injection gates for the experiments. The procedure is explained in detail in the next chapter.

4.4.2 Circuit Augmentation

To run fault-injection experiments, the tested circuit needs to be augmented with fault-injection inputs. When one of these fault-injection inputs gets asserted, it causes a fault on one of the wires in the circuit. While the framework also supports using a prepared circuit which already provides those inputs, one of its main advantages is the possibility of taking an arbitrary QDI circuit in PRS form and automatically augmenting the circuit with the fault-injection inputs. This is realized in the `Augmenter` Python class.

The fault-injection gates are added as inputs to the PRS and then selected assignments (the victims) are changed by rewiring their targets. Instead of the original target they go to temporary wires which are used as inputs to the gates that perform the actual fault injection. Three different kinds of faults are supported by the framework: Bit flips, Stuck-at-0 and Stuck-at-1. For bit flips the value of a wire is inverted while for stuck-at-0 and stuck-at-1 the value is set to a constant 0 or 1 respectively. To achieve these effects XOR, AND or OR gates are used accordingly. A simple example of an assignment being augmented is shown in Listing 4.3 (for simplicity reasons the example is not an actual QDI circuit).

```
prs example is
inputs
  a : Bit;
  b : Bit;
outputs
  x : Bit;
begin
  x := and_gate(a, b);
end prs;
```

Listing 4.2: Example circuit before augmentation

```
prs example is
inputs
  a : Bit;
  b : Bit;
  err : Bit(1);
outputs
  x : Bit;
begin
  x_pre_err := and_gate(a, b);
  x := xor_gate(x_pre_err, err(0))
end prs;
```

Listing 4.3: Example circuit after adding a bit flip injection gate.

One key feature of the PRS format in `pypr` are attributes that can be added to signals to provide further information about their functionality. One common use case of attributes is for grouping together signals that belong to particular channels. In the case of circuit augmentation, the fault-injection inputs in the resulting PRS are also extended with attributes to allow for automatic processing in subsequent scripts. An example is shown in Listing 4.4. It shows the declaration of a multiplier circuit where the two input vectors together constitute an input channel and the result vector an output channel.

```
prs umul is
inputs
  a : DRBit(4) attributes(channel := Cin, role := data);
  b : DRBit(4) attributes(channel := Cin, role := data);
  ack_in : Bit attributes(channel := Cout, role := ack, channel_type := DIDR);
  reset : Bit attributes(role := reset);
  err_inj : Bit(32) attributes(role := errinj, type := FLIP);
outputs
  ack_out : Bit attributes(channel := Cin, role := ack, channel_type := DIDR);
  s : DRBit(8) attributes(channel := Cout, role := data);
```

Listing 4.4: PRS header of a multiplier circuit that has been augmented with 32 bit-flip injection gates.

When it comes to choosing victims (i.e., wires that are augmented with injection gates), the library generally performs a random approach for choosing the signals. The only exception are assignments that directly connect to the output channel. These are always skipped as an fault injection at the output would no longer test the fault robustness of the circuit itself. Additionally, it is possible to pass a regular expression to pre-filter what wires are potential victims. The `Augmenter` class accepts the number of injection gates to add, the type of fault, a regular expression for pre-filtering victim wires and a seed that is used for the random selection of victims as parameters. Additionally a counter can be passed to discard the first iterations of the chosen gates. This allows randomly iterating over all gates in the circuit without repetition by repeatedly augmenting the circuit with the same seed and only incrementing this counter.

It should be noted that it is also possible to add fault-injection inputs for all rules at once, thus only needing one synthesis run to test all gates. One downside of this approach is that every injection gate adds a certain delay, thus changing the timing of the circuit. While the placement on the FPGA will always be different between synthesis runs, the timing of the circuit should be closer to its original implementation when keeping the number of injection gates low.

4.4.3 Circuit Wrapping

After having obtained a PRS description of an augmented circuit (either manually or by using the `Augmenter` class), the PRS code still needs to be converted into a VHDL representation that fits with the framework design to actually be synthesizable. For these purposes, the Python library provides a `Wrapper` class, working in two steps.

First it creates a PRS wrapper around the PRS circuit which follows the exact specification of the entity that is required by the framework. This includes making sure that the input and output channels are joined together in one vector each and that the entity and all its ports have the expected names.

In the second step, the actual files are created out of the PRS wrapper for further use in the framework. The circuit is exported into two VHDL entities, where one is intended for the actual FPGA implementation, meaning that all gates are mapped to specific VHDL components, ensuring that the correct mapping to the FPGA's hardware resources is used. This generation of the VHDL entity also allows the addition of delay gates to artificially slow down the UUT. The gates will be added at the circuit inputs and at the outputs of all logic gates in the circuit to slow down the circuit as evenly as possible. The other VHDL entity generated is for simulation purposes and makes use of VHDL processes to represent the individual gates. The simulation entity is primarily intended for debugging purposes and has no impact when only running hardware injections.

Additionally, a VHDL package is generated which contains generics about the tested circuit like port widths, maximum sequence lengths or the number of injection gates. The last file that is generated is a Python package called `UUT.py` which contains those same constants and some more if they are available (e.g., a list of names that correspond to the victim gates). The Python package is used by the rest of the library to send the right commands to the framework and for being able to generate reproducible and useful results.

When generating the files, the output directories for the VHDL and Python files need to be provided as well as the maximum sequence length that should be allowed to be written as input sequence. As part of the framework, a script is provided (generate_uut_instance_from_prs.py) which can be used to take an existing PRS file, augment it with fault-injection gates, and generate the VHDL and Python files that are required for synthesizing the fault-injection framework and running experiments.

4.5 Experiment Control Software

This section describes the Python library components that have been developed for controlling the injection harness by configuring the execution parameters, starting the experiments and processing the results.

It also comes with a class for executing large-scale experiments on specific circuits and storing the results. The results of experiments can be parsed by a result manager to filter for certain properties and to create plots from the data to simplify post-processing and analysis.

4.5.1 UART Driver

The `UartDriver` class is the abstraction layer between the FPGA and the actual experiment execution by providing functions for configuring the tester, starting runs and receiving results. It is extensively used by the other classes in the framework for test execution.

When writing an input sequence using this class, the input words are generated randomly, but a seed can be passed. This way it is guaranteed that experiments can be reliably reproduced.

4.5.2 Experiment Execution

For running automated large-scale experiments on circuits the `Executor` class can be used. It takes an experiment configuration and automatically performs all the requested fault injections, storing the results in Comma Separated Values (CSV) files for further analysis.

The experiment configuration is provided in the JSON format, accepting a number of different options for covering as much of the parameter space as possible. Such a JSON example is shown in Listing 4.5. It demonstrates how to iterate over different parameters with one configuration file. An example are the `logicStyle` and `bufferStyle` parameters which are directly incorporated as placeholders in the filename, enabling experiment execution over multiple input files with just one experiment configuration.

In some cases the value -1 is used to iterate over all possible combinations. In the case of `numPulseStarts` being -1, a golden run is executed before the actual test execution for finding the ideal value for `numPulseStarts` to inject an error at every possible point in time until the last token has been received. The `runPart` parameter is passed to the `Augmenter` to choose which set of victims to take. When it is set to -1, all sets will be iterated over until all injection gates have been tested.

```json
{
    "name" : "lfsr16b",
    "logicStyle" : ["NCLX", "DIMS"],
    "bufferStyle" : ["WCHB", "DeadlockingWCHB", "DualCDWCHB"],
    "file" : "prs/lfsr_{bufferStyle}_{logicStyle}.prs",
    "resultDir" : "./results/",
    "skipReruns" : 0,
    "runType" : "Augment",
    "faultType" : "FLIP",
    "gatesPerRun" : 64,
    "seqLength" : 32,
    "runPart" : -1,
    "testParams" : {
        "minPulseWidth"   : 1,
        "incPulseWidth"   : 1,
        "numPulseWidths"  : 20,
        "minPulseStart"   : 0,
        "incPulseStart"   : 1,
        "numPulseStarts"  : -1,
        "expectedOutputs" : 32,
        "outputDelay"     : [0,1,2],
        "inputDelay"      : 0
    },
    "seed" : 74,
    "uart" : {
        "port" : "/dev/ttyUSB0",
        "baud" : 576000
    }
}
```

Listing 4.5: Example of an experiment configuration in JSON format.

When the `Executor` is run on a configuration it will iterate over the full parameter space, perform the still required operations (e.g., augmentation, wrapping, synthesis, flashing) and then run the actual fault-injection experiments as specified by the test parameters. The results are collected and then passed on to the results manager (see Chapter 4.5.3) where they get directly exported to the associated CSV files.

4.5.3 Result Manager

To keep track of experiment outcomes the `ResultManager` class has been developed. It uses CSV files for importing/exporting the result data and pandas[2] dataframes for managing the datasets internally. The class uses two datasets to manage the results.

In the 'runs' dataset, there is an entry for each individual fault-injection experiment, providing a summarized view of the injected faults and the resulting observations. It contains the following information:

[2] https://pandas.pydata.org/

- `name`: Arbitrary name chosen for the experiment as written in the experiment configuration
- `file`: Path of the PRS file that was used as input
- `seed`: Seed used for choosing victim gates and also the input sequence
- `logicStyle`: Logic style used on the circuit (e.g., DIMS, NCLX)
- `bufferStyle`: Buffer style used (e.g. WCHB)
- `faultType`: Fault type (i.e., FLIP, SA0 or SA1)
- `minPulseStart`, `incPulseStart`, `numPulseStarts`, `minPulseWidth`, `incPulseWidth`, `numPulseWidths`: Information about which pulses were tested exactly
- `inputDelay`, `outputDelay`: Delays (in core clock cycles) that were applied when generating/acknowledging the input and output tokens
- `numGates`: Number of fault-injection gates
- `totalRuns`: Number of individual fault injections performed (product of `numGates`, `numPulseWidths` and `numPulseStarts`)
- `valueErrors`, `glitchErrors`, `codeErrors`, `deadlocks`, `countErrors`, `timingDeviations`: The number of times that each error has been observed.
- `anyErrors`: The number of times that at least one error was observed (does not include timing deviations)
- `anyDeviations`: The number of times that any deviation was observed
- `multiErrors`: The number of times that multiple errors were observed at once (does not include timing deviations)
- `victimGates`: List of gates that were chosen for the fault injections. The indexes of the list correspond to the gate IDs seen in the 'results' dataframe

The 'results' dataset contains an entry for each individual fault injection that caused a deviation or was the golden run of a specific run. As one would expect, this dataset can get very big containing millions of entries. It is useful when detailed analysis of results is required or the effect of faults on particular gates is of interest. The stored data fields are:

- `runId`: Link between the 'runs' and 'results' dataframe, making it possible to trace back to the run that produced the result

- `duration`: Duration that the run took in nanoseconds. The duration is always until the last output token was observed (i.e., for a deadlock the timeout is not included)
- `faultStart`: Time in nanoseconds at which the fault was injected, counted from when the reset is removed from the UUT.
- `faultDuration`: Time in nanoseconds for which the fault was applied.
- `faultGateId`: ID of the fault-injection gate used.
- `faultGateName`: Name of the gate corresponding to the gate ID. This only has a valid value if `UUT.py` contains the appropriate information (done automatically when the full augmentation flow of the library is used)
- `faultType`: Type of fault: FLIP, SA0 or SA1
- `valueError`, `glitchError`, `codeError`, `deadlock`, `countError`, `timingDeviation`: Boolean value indicating if the respective error has been observed
- `anyDeviation`: Boolean value indicating whether any deviation has been observed
- `anyError`: Boolean value indicating whether any error has been observed (excludes timing deviations)
- `multiError`: Boolean value indicating whether multiple errors have been observed (excludes timing deviations)

The class provides functions for filtering by fields and summing up multiple runs for easy processing of the data, simplifying the consolidation of data for plotting and further analysis.

4.6 FPGA Integration

The full framework design has been ported to the DE2-115 Development and Education Board[3] for running experiments on real hardware. The board comes equipped with the Cyclone IV EP4CE115 FPGA and an integrated RS232 serial port used for communication with the host PC. Some of the LEDs on the board are additionally used to provide visual feedback of the current state of the framework.

As the framework itself is mostly self-contained, integrating it into a platform is not very difficult. The primary task is generating the clocks that are used to drive the entire design, i.e., the core clock and the injection clock. As noted previously, both clocks must be derived from the same source and the injection clock frequency must be a multiple of

[3]`https://de2-115.terasic.com/`

the core clock frequency (i.e., they can also be the same). The second step is instantiating the framework itself with appropriate generic settings (i.e., UART settings and clock frequencies) and connecting the pins correctly.

To make sure the asynchronous circuits are really instantiated as intended it is also necessary to provide declarations of basic logic gates (e.g., C Gates, OR-Gates, . . .) which should ideally each be bound to exactly one LUT on the device. The PRS conversion generates exactly these basic gates which are then instantiated at synthesis. For the development board used, Altera primitives are directly used in these gate declarations to directly map to the LUTs of the FPGA. This means, that the implementation specific source code files can be reused if someone intends to port the design to another Altera/Intel FPGA. If a different manufacturer is targeted (e.g., Xilinx), these gates will need to be re-implemented for that specific platform.

An overview of all parameters used in the implementation used for the experiments in this thesis is given in Table 4.3.

Parameter	**Value**
Platform	DE2-115
FPGA	EP4CE115
Core clock frequency	100 MHz
Injection clock frequency	200 MHz
Baud Rate	576,000
Parity	Even
Stop Bits	1

Table 4.3: Implementation Parameters

CHAPTER 5

Experiment Configurations

The framework is evaluated by running automated fault-injection experiments on multiple circuits. All circuits will be generated with various implementation techniques (namely different logic and buffer styles) and the results will be compared. This chapter provides a detailed description of the exact configurations used throughout this thesis. Chapter 5.1 will functionally describe the QDI circuits while Chapter 5.2 will go into detail on the different buffer styles with which those circuits will be generated. In addition to the buffer styles, the experiments will be tested with two of the logic styles that have been presented earlier: Delay-Insensitive Minterm Synthesis (DIMS) and Null Convention Logic with Explicit Completion (NCLX).

5.1 Test Circuits

5.1.1 Linear-Feedback Shift Register

The Linear-Feedback Shift Register (LFSR) is a circuit where a register of fixed length is continuously shifted by one bit, with the lowest bit being removed and the highest bit getting set according to a linear function based on its previous state. The most common replacement strategy for the highest bit is an XOR function fed with specific bits of the state register. The feedback function is most commonly expressed as a polynomial where the degrees of the individual terms represent the bits to be used for the XOR function. A simple LFSR example is shown in Figure 5.1.

For the experiments, a purely combinational 16 bit LFSR module (using $x^{16} + x^{15} + x^{13} + x^4 + 1$ as polynomial) was implemented in Verilog which performs 16 steps at a time. By doing 16 steps at once it is ensured that all bits need to be generated by some actual logic instead of just shifting them to a new position.

The actual circuit under test is comprised of a ring of three buffers and the combinational LFSR logic as depicted in Figure 5.2. Additionally, there is a fork between buffers B_1 and

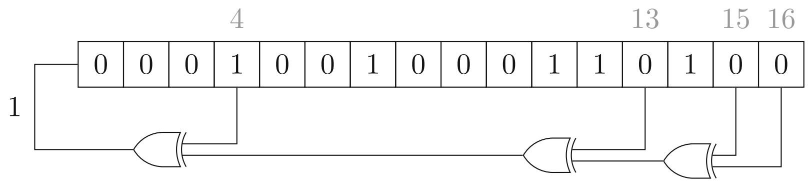


Figure 5.1: 16-bit LFSR example using the feedback polynomial $x^{16} + x^{15} + x^{13} + x^4 + 1$. The register's current value is 0x1234 and will be 0x891A in the next step.

B_2, with the second output being directly connected to the module's output. The buffer in the middle (B_1) is initialized with a data token (in our case 0x1234), meaning that once the reset is removed, the data token is immediately provided to the output as well as to B_2. As long as the output tokens get acknowledged, the circuit will continuously generate new data without the need for an input channel. The DR implementations of all the components are provided by `pypr`, which also takes care of synthesizing the combinational LFSR module either in the NCLX or the DIMS logic style.

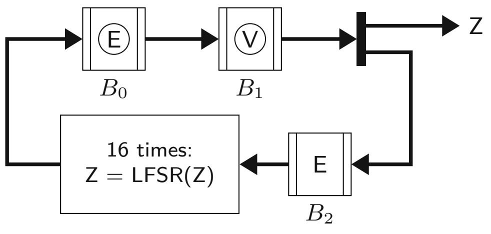


Figure 5.2: SDFS diagram of the tested LFSR circuit in its initial state.

The circuits are generated with two logic types and six different buffer types as presented in Chapter 5.2 leading to a total of 12 different implementations that will be compared with the framework.

5.1.2 Carry-Ripple Adder

The last circuit we are going to look at is the 4-bit carry-ripple adder that was analyzed in [MRL04], see also Chapter 3.1. It is built from four full adders with carry bits whose design is shown in Figure 5.4. The full design is depicted in Figure 5.3.

It can be seen that the *Cout* signal in the individual full adders is independent from *Cin* if A and B are both set to the same value. Thus, the four carry signals can be calculated independently from each other if A equals B, constituting the best case scenario. On the other hand, whenever A and B differ in all four bits, the *Cout* output of each adder depends on the *Cin* input coming from the previous instance, thus activating the critical path of the circuit. Because of these different behaviors, the circuit will be subjected to different input sets to compare the impact this has on the fault sensitivity.

As [MRL04] was analyzing the fault sensitivity of the individual C gates in the design, the experiments will focus on those as well. For this purpose, a fault-injection gate is inserted at every C gate input in the design (indicated by the lightning bolts in Figure 5.4). This positioning of the injection gates ensures that every fault can only directly affect one C gate. As there are 10 C gates with two inputs each, there is a total of 80 fault-injection gates to be inserted over all four full adders. To affect the inter-circuit timing as little as possible, all injection gates will be added at once when running the experiments, essentially slowing down all C gates roughly by the same amount.

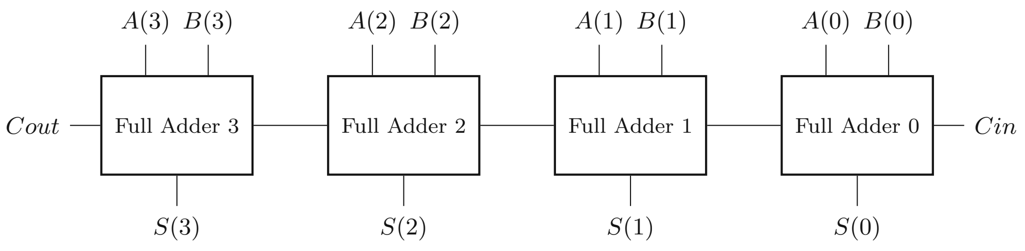


Figure 5.3: Schematic of the full 4-bit adder. All the signals are DR.

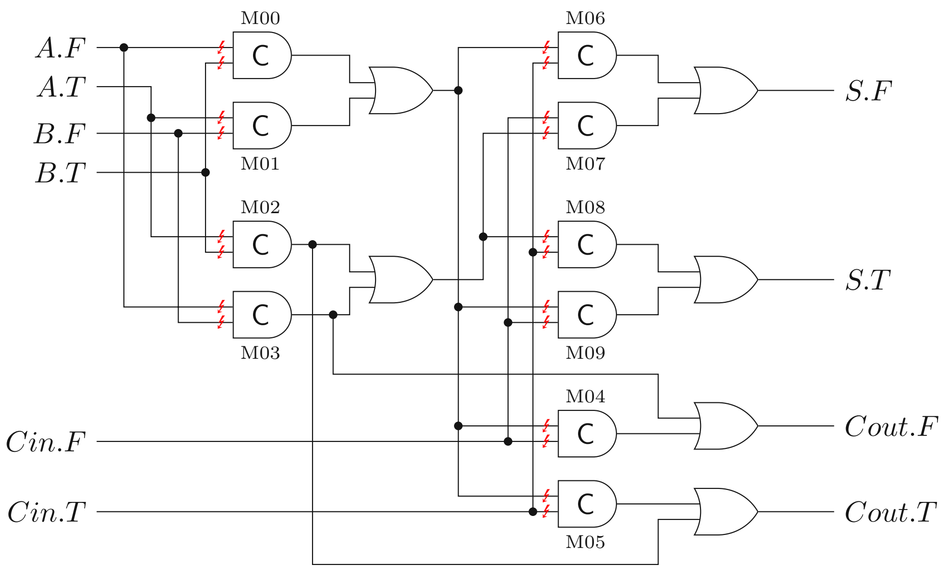


Figure 5.4: Schematic of the combinational full adder. The lightning bolts indicate the positions where fault-injection gates are added.

5.1.3 Multiplier

The second circuit to be analyzed is an unsigned N-bit multiplier. The design takes two N-bit numbers and generates the $N \times 2$-bit product as output. It is implemented as a pipeline with N stages for maximum throughput. It consists of N data buffers (D_0 to D_{N-1}) and $N-1$ logic blocks that calculate the product $A \times B$ bit by bit. The structure of the circuit for $N=4$ is shown in Figure 5.5. The required combinational logic of the individual stages is mostly comprised of full adders which are implemented either using the DIMS or the NCLX logic style.

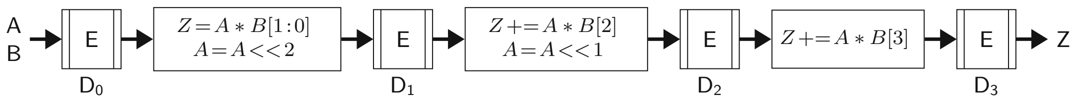


Figure 5.5: 4-bit SDFS example of the tested multiplier. The first logic block processes two bits at once because there is no Z yet, reducing the initial complexity.

As the multiplier is implemented in the form of a pipeline, it is possible that multiple products are calculated in an interleaved fashion. To verify the impact of different fill levels of the pipeline, the experiments will be executed with different input and output delay settings.

5.2 Buffer Styles

The most basic buffer style to be tested with the framework is the WCHB as presented in Section 2.5.1. Whenever the WCHB is awaiting a new data word, the input C gates are in a 1-sensitive to 0 state, leaving a potentially large time window during which the structure is sensitive to faults. To mitigate these issues, various buffer styles have been devised, aimed at reducing different kinds of errors. Many of these styles are based on the WCHB. The buffer types that are evaluated with the framework will be referred to according to the names given in Table 5.1.

Buffer Type	**Short Name**	**Details**
Weak-Conditioned Half Buffer	WCHB	Chapter 2.5.1
Deadlocking WCHB	Deadlocking	Chapter 5.2.1
Interlocking WCHB	Interlocking	Chapter 5.2.2
Dual Completion Detector WCHB	DualCD	Chapter 5.2.3
[BS09] type Locking WCHB	Locking	Chapter 5.2.4
MOUSETRAP-Style D-Latch Half Buffer	Mousetrap	Chapter 5.2.5

Table 5.1: Tested buffer styles

5.2.1 Deadlocking WCHB

The Deadlocking WCHB is a modified version of the WCHB where asymmetric C gates are used for storing the data. Asymmetric C gates provide additional inputs which are only considered for their respective transitions, i.e., positive inputs need to be high for a transition to 1 while negative inputs need to be low for a transition to 0. These inputs are not considered for the opposite transition. In the case of the Deadlocking WCHB, asymmetric C gates with negative inputs are used. The outputs of the C gates storing the DR data are connected to the negative inputs of the opposite rail as depicted in Figure 5.6.

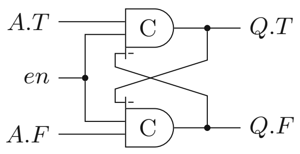


Figure 5.6: Deadlocking Buffer. An invalid word will cause a deadlock that can only be resolved with a reset.

The feedback of the output to the negative inputs of the C gates ensures that the circuit immediately deadlocks if an invalid DR word is captured (i.e., both data rails set) by preventing the buffer from entering the null phase ever again. This buffer type is well suited in cases where value errors are unacceptable while deadlocks can be handled (e.g., safety critical systems with redundancy).

5.2.2 Interlocking WCHB

Similarly to the Deadlocking WCHB, the interlocking variant also uses cross-coupled asymmetric C gates. In this buffer style, the feedback signal is inverted and fed into the positive input of the asymmetric C gate as depicted in Figure 5.7.

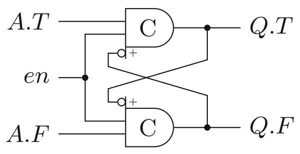


Figure 5.7: Interlocking Buffer. Once a valid state has been captured, a transition on the other input will be ignored.

With this modification, only the first transition is propagated to the output as the feedback line effectively disables the other C gate from performing a rising transition. This structure is especially useful for avoiding code errors. On the other hand, the likeliness of a fault causing a value error is significantly higher as the fault can cause an immediate propagation of a wrong value. While the likeliness of a code error is significantly reduced, the possibility is not entirely eliminated as both transitions could arrive so close to each other that the feedback signals did not have enough time to propagate back to the inputs.

5.2.3 Dual Completion Detector WCHB

Another possibility to reduce the window in which the C gates are sensitive to faults is an additional CD in front of the WCHB to only arm the C gates once all inputs are either in the data or the spacer phase, see also Figure 5.8. The Dual-CD WCHB masks transient faults that occur while the next token is not completely processed yet. If a fault occurs after the completion detector has already switched, it might still be propagated.

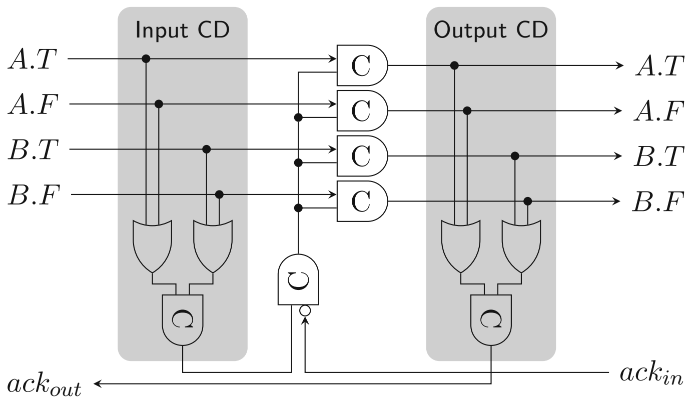


Figure 5.8: Example of a Dual-CD WCHB storing two DR bits. The C gates get armed for the next transition only when both bits have finished their transitions.

5.2.4 Locking WCHB

Another variant of the WCHB has been proposed in [BS09] and is shown in Figure 5.9. This version can be seen as a generalized version of the Interlocking WCHB by using the output of the CD for locking the C gates instead of directly connecting to the opposite data rail. Thus, this variant can also be used for non-DR protocols (i.e., 1-of-N).

In our implementation, the individual CDs of the bits are connected directly to the C gates to minimize the sensitivity windows as much as possible. This behavior is not clearly specified in [BS09], another option could be to use the overall CD output to lock all C gates at once, i.e. locking all C gates as soon as the full token has been stored.

When using the individual CD outputs, it is functionally identical to the Interlocking WCHB, but we would expect it to be slightly less effective as the locking signal goes through an additional logic gate before reaching the asynchronous inputs of the C gates, slightly delaying the locking behavior.

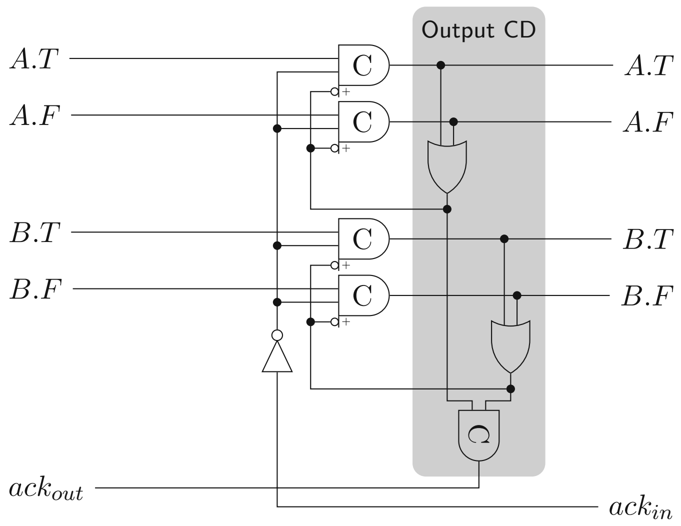


Figure 5.9: Example of the Locking WCHB based on [BS09] storing two DR bits. The C gates of each bit are locked as soon as one of the data rails goes high.

5.2.5 MOUSETRAP-Style D-Latch Half Buffer (MDHB)

The last buffer style to be tested is the MOUSETRAP-Style D-Latch Half Buffer (MDHB) which is a completely different approach based on the MOUSETRAP pipeline style introduced in [SN07]. In this style, D-Latches are used instead of C gates for storing the data. As seen in Figure 5.10, the enable inputs of the D-Latches are controlled by an XNOR gate using the acknowledgement from the next stage and the output of the CD. In this setup, the latches become transparent for the data when the next stage acknowledges the current output and close as soon as the next phase has been detected by the CD.

It needs to be noted that this technique introduces a small timing constraint between the output of the completion detector and the acknowledgement from the next stage and as such is not strictly QDI. Furthermore, this structure is not meant to improve fault robustness over the other buffer types. It is included in the comparison to see how a D-Latch based approach fares compared to the C-gate-based buffers.

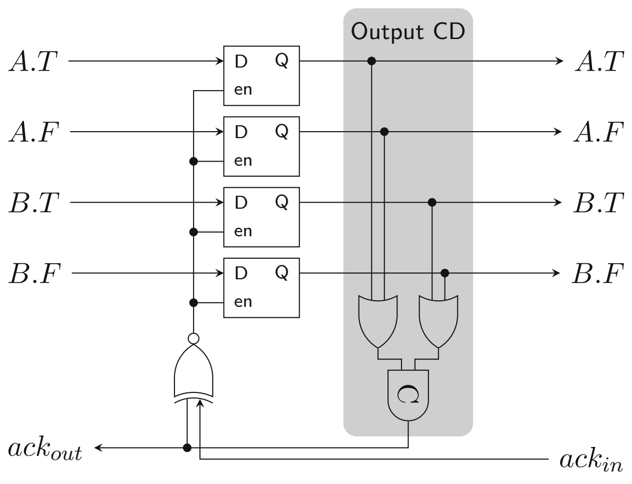


Figure 5.10: Example implementation of a MDHB storing two DR bits.

CHAPTER 6

Experiment Results

This chapter presents and analyses the observations made while running fault-injection experiments on the circuits introduced in the previous chapter. We mostly focus on statistical analysis of the observed effects but also provide some insights into the performance of the framework itself.

When analyzing the individual error rates, any occurrence of an error is counted, independently of whether it was the only one observed in a run. Thus, the overall error rate may sometimes be lower than the sum of the individual error rates that are presented. One exception from this rule is how deadlocks are counted: A deadlock is only counted as such if the same injection did not also result in a code error or glitch. This is to ensure that the deadlocked state is reached internally and not simply because an invalid output token is not acknowledged by the completion detector at the output.

6.1 LFSR

The LFSR circuit was tested with the following parameters:

- All six buffer types
- Both logic styles (DIMS and NCLX)
- Three fault types: Bit flips, stuck-at-0 and stuck-at-1
- Output delays of 0, 10 and 20 ns
- 10 different fault durations, from 5 to 50 ns
- All possible fault start times within the range of golden run execution
- 32 fault-injection gates at a time

6.1.1 Encountered Problems

During execution there were problems with the MDHB in both logic styles. The experiments could not always be executed properly because the golden run was unable to finish, indicating some problem in the generated circuit. The cause for this problem could not be found. In total, this caused around 20% of runs to fail in the DIMS logic style and around 15% in the NCLX logic style. Because of this, the numbers presented for MDHB might not be fully representative as the injection experiments were not executed on all potential victims.

6.1.2 Error Rates

The error rates for value errors, deadlocks, code errors and glitches are presented in Figure 6.1. Additionally, the overall error rate is shown. Count errors are not included because they are not reported in circuits without input channels. The plots combine the error rates for different output delays, as the impact of the parameter was negligible on the LFSR circuit.

It can be seen that almost all error types as well as the overall error rate are more common in the DIMS logic style. A likely cause for this effect is the reliance of DIMS on C gates. The 16-bit combinational LFSR logic in DIMS style uses 144 C gates, resulting in roughly 50% more C gates than the NCLX implementation. As each C gate has the possibility of capturing a fault, turning it from an SET to an SEU, the increased error rate makes perfect sense.

Another observation to be made is that the **deadlocking WCHB** seems to work as intended. The (already low) rate of value errors and glitches is noticeably lower than with the standard WCHB implementation while deadlocks are slightly more common. The frequency of code errors does not change much, which is easy to explain as an individual stage that deadlocks will still store the code error and potentially propagate it to the output. This also explains why the rate of deadlocks does not see a significant spike as the deadlocking usually coincides with a code error and is only counted as such.

The **interlocking and locking WCHB** implementations behave very similarly to each other, producing the lowest overall error rates which mostly stems from the reduction in code errors. This is the expected behavior because the individual bits lock in on the first transition of a data rail, thus, filtering out faults on the other data rail that occur after the buffer has been locked. This, of course, also works the other way around. If a fault on the wrong data rail arrives first, the wrong bit will be locked in and the valid transition gets ignored. This is clearly visible in the vastly increased rate of value errors. An interesting observation is the increase in deadlocks which suggests that while these buffers are effective at reducing code errors at the output, the faults still cause problems internally which lead to a deadlock. It seems likely that some of the internal DR buffers still reach an invalid state which can happen when the two transitions arrive close enough to each other. This theory is supported by the locking WCHB showing the

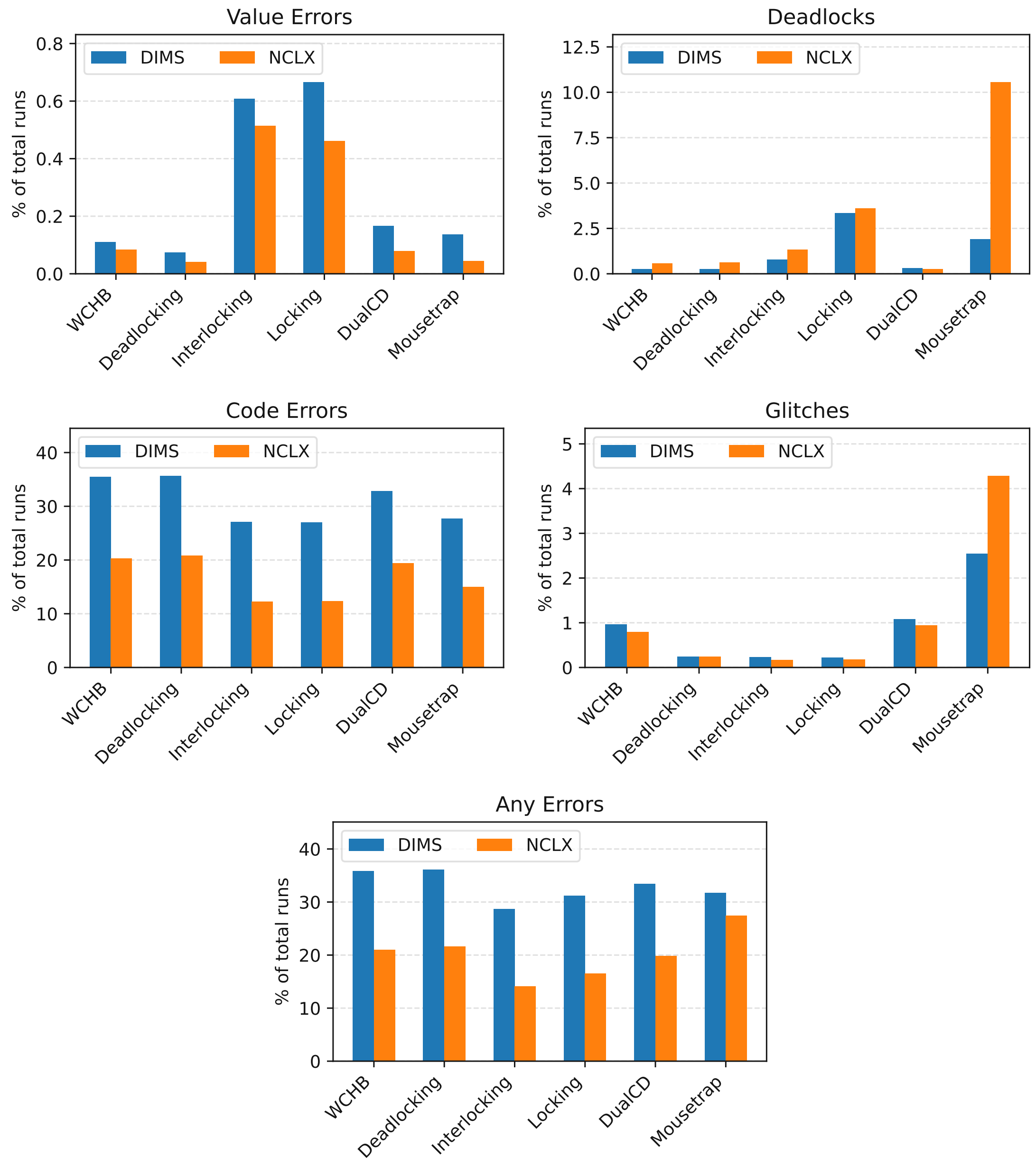


Figure 6.1: LFSR error rates relative to all performed bit flip injections

higher deadlocking rate as it has the longer path to lock the other C gate, resulting in a bigger sensitivity window.

Perhaps most interesting about the **MDHB** results is the high number of glitches compared to all other buffer types. This makes sense when considering the transparent latch behavior when the next stage has acknowledged the last output. Any faulty

Buffer Style	Value Errors	Glitches	Code Errors	Deadlocks	Any Errors
WCHB	−0.02%	+0.02%	+0.02%	−0.20%	+0.02%
Deadlocking	−0.45%	−0.29%	+0.02%	−0.10%	+0.02%
Interlocking	−0.09%	+0.16%	+0.01%	−0.11%	+0.01%
Locking	+0.49%	+0.06%	+0.02%	+0.15%	+0.06%
DualCD	−0.20%	+0.14%	+0.01%	−0.08%	+0.01%
Mousetrap	−0.95%	+0.08%	−0.01%	+0.07%	+0.01%

Table 6.1: Error rates when rerunning the LFSR experiment on the same board, relative to the original run

Buffer Style	Value Errors	Glitches	Code Errors	Deadlocks	Any Errors
WCHB	+0.96%	−0.47%	−0.58%	+0.88%	−0.54%
Deadlocking	+6.24%	−5.12%	−0.07%	+0.64%	−0.14%
Interlocking	+2.92%	−2.71%	−0.80%	−0.09%	−0.77%
Locking	+1.38%	−6.65%	−0.36%	+0.06%	−0.41%
DualCD	+7.55%	−0.24%	−0.31%	+0.45%	−0.28%
Mousetrap	+7.47%	−4.92%	+2.84%	−6.62%	+0.23%

Table 6.2: Error rates when rerunning the LFSR experiment on a second board, relative to the original run

transitions during that phase will be propagated as-is including the dissipation of the fault if the completion detector does not lock the latches in between. This differentiates the buffer style from C gate based implementations where the C gates generally store the observed value until the next data phase. The other interesting point is the spike in deadlocks in the NCLX logic style. The reason for this behavior was not further analyzed but it might also be related to C gates as the DIMS logic style still uses them in the combinational logic while they are completely absent in NCLX.

6.1.3 Reproducibility

One interesting aspect in evaluating the framework is the reproducibility of results. For that purpose, the LFSR experiments have been run three times in total to compare the error rates. It was run twice on the same board and then an additional time on a different board to investigate how different hardware can affect the error rates.

The relative error rates of the rerun on the same boards are listed in Table 6.1, showing only the relative change compared to the results seen in Figure 6.1. It is clearly visible that the error rates only change marginally, with the changes mostly staying under 0.1% and never exceeding 1%. This demonstrates that the framework is generally able to reproduce results and behaves as expected. Small deviations can unfortunately not be avoided as asynchronous circuits are inherently influenced by external factors like junction temperature which are impossible to control completely.

The same statistics when running the experiment on a second board are shown in Table 6.2. In this case, the differences are clearly much more pronounced, especially when looking at the less common value errors and glitches. While the differences seem to be quite high at first glance, it is important to put these numbers into context: A 7.55% increase of value errors with the DualCD buffer corresponds to the error rate rising from 0.1182% to 0.1272%, showing that the actual number of observed errors is still very similar when running on the same board. More importantly, the overall trends that can be observed on the first board also hold for the second board.

6.2 Multiplier

The multiplier circuit was tested in a 4-bit and an 8-bit configuration with this set of parameters:

- All six buffer types
- Both logic styles (DIMS and NCLX)
- One fault type: Bit flips
- Input delays of 0 and 10 ns
- Output delays of 0, 10 and 20 ns
- 10 different fault durations, from 5 to 50 ns
- All possible fault start times within the range of golden run execution
- 32 fault-injection gates at a time (128 for the 8-bit version)

The input data that is used for the experiments is a sequence of 32 pseudo-randomized tokens, being the same for every execution of the test framework to ensure that the processed data is always the same.

The general error rates for the 4-bit configuration are depicted in Figure 6.2 and show a similar picture to what was observed with the LFSR circuit. No count errors were observed (neither on the 4-bit nor on the 8-bit implementation). This is no surprise because the odds of a single SET to generate an additional data token are extremely low.

When comparing these results to what was observed on the simulation framework in [Beh21], the overall trends generally match, even though the percentages are usually in different ranges. The most likely reason for this is the decreased level of control over the circuit in hardware, e.g., not being able to perform injections that last less than 5 ns or having slightly skewed timing due to the synchronizers.

Compared to the LFSR results, an interesting observation is the rate of value errors which is higher by almost an order of magnitude in most of the instances, for the interlocking and locking buffer styles exceeding 6% consistently. This suggests that the multiplier circuit is relatively sensitive to producing wrong outputs when faced with SETs compared to the LFSR. This is not too surprising given that the multiplier circuit contains a lot more combinational logic. While many faults in the LFSR circuit will directly go into C gates, most likely causing a code error or no error at all (if the C gate was expected to toggle), the faults in the multiplier circuit are much more likely to affect wires in the combinational logic, where the effect is not as straightforward to predict. Even faults that are directly captured by C gates as code errors will most of the time still need to pass through a few stages of combinational logic. The logic blocks themselves do not define a specific behavior for erroneous inputs and as such, might not (or with delay) propagate the code error to their outputs, but instead (temporarily) provide a valid data word with a wrong value that is captured by the next buffer, providing another potential reason for the increase in value errors.

Most of the other errors occur at a slightly lower rate except for glitches. Similarly to the value errors, this can be explained by the more complex combinational logic, making it more likely that a fault does not directly get propagated by a C gate or simply dissipates before it gets captured.

When looking at the error rates of the 8-bit configuration in Figure 6.3, it can be seen that the data width of the multiplier does not have a noticeable impact on the overall error rate for any buffer type. The interlocking and locking WCHB produce even more value errors, which indicates that the longer data paths manage to filter out even more code errors as one would expect. The other significant change is the reduction of glitches which is also consistent with the expected behavior, because the added logic makes it even more unlikely for glitches to occur.

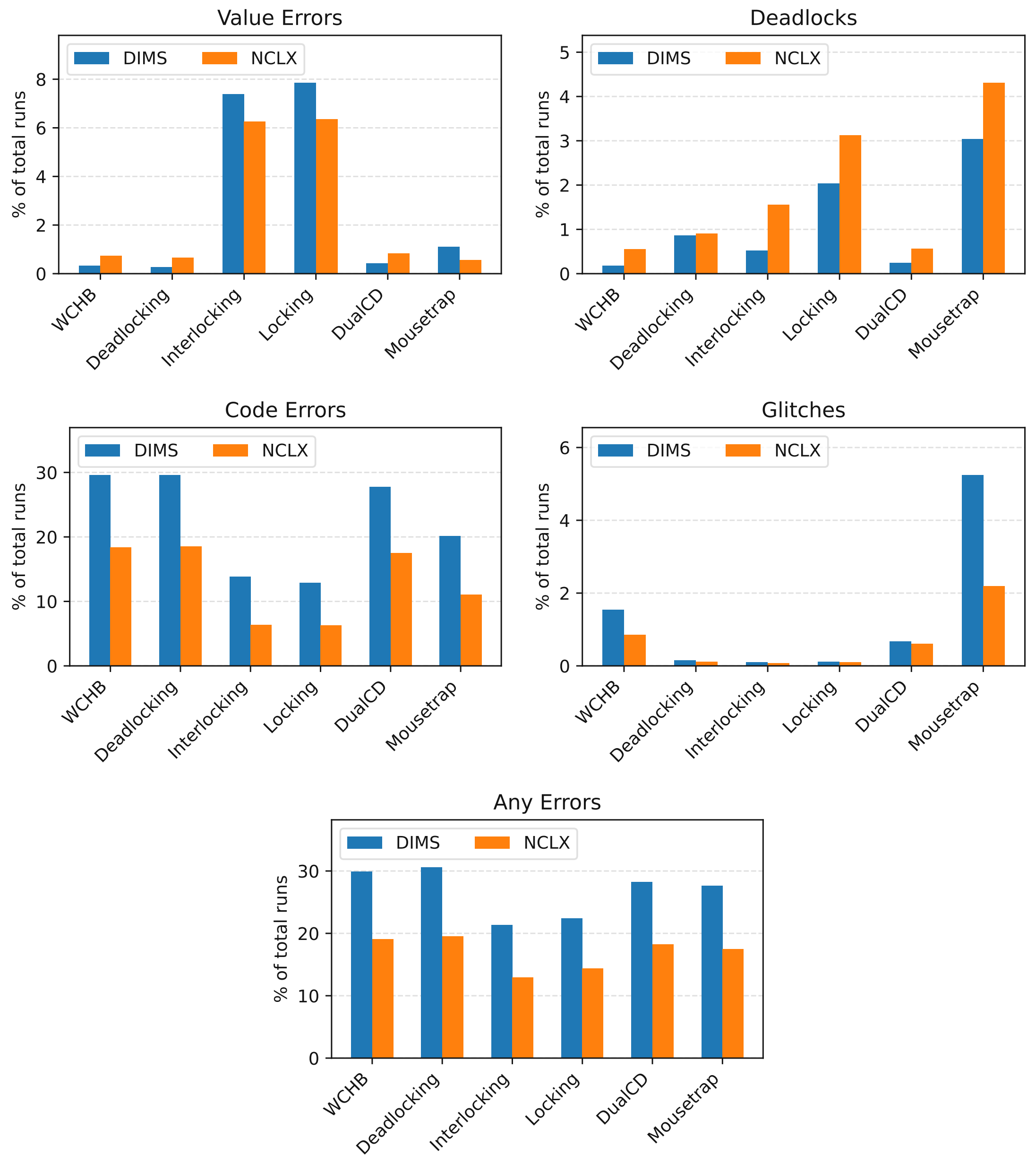


Figure 6.2: 4-bit Multiplier error rates of the different buffer types and logic styles relative to all performed bit flip injections

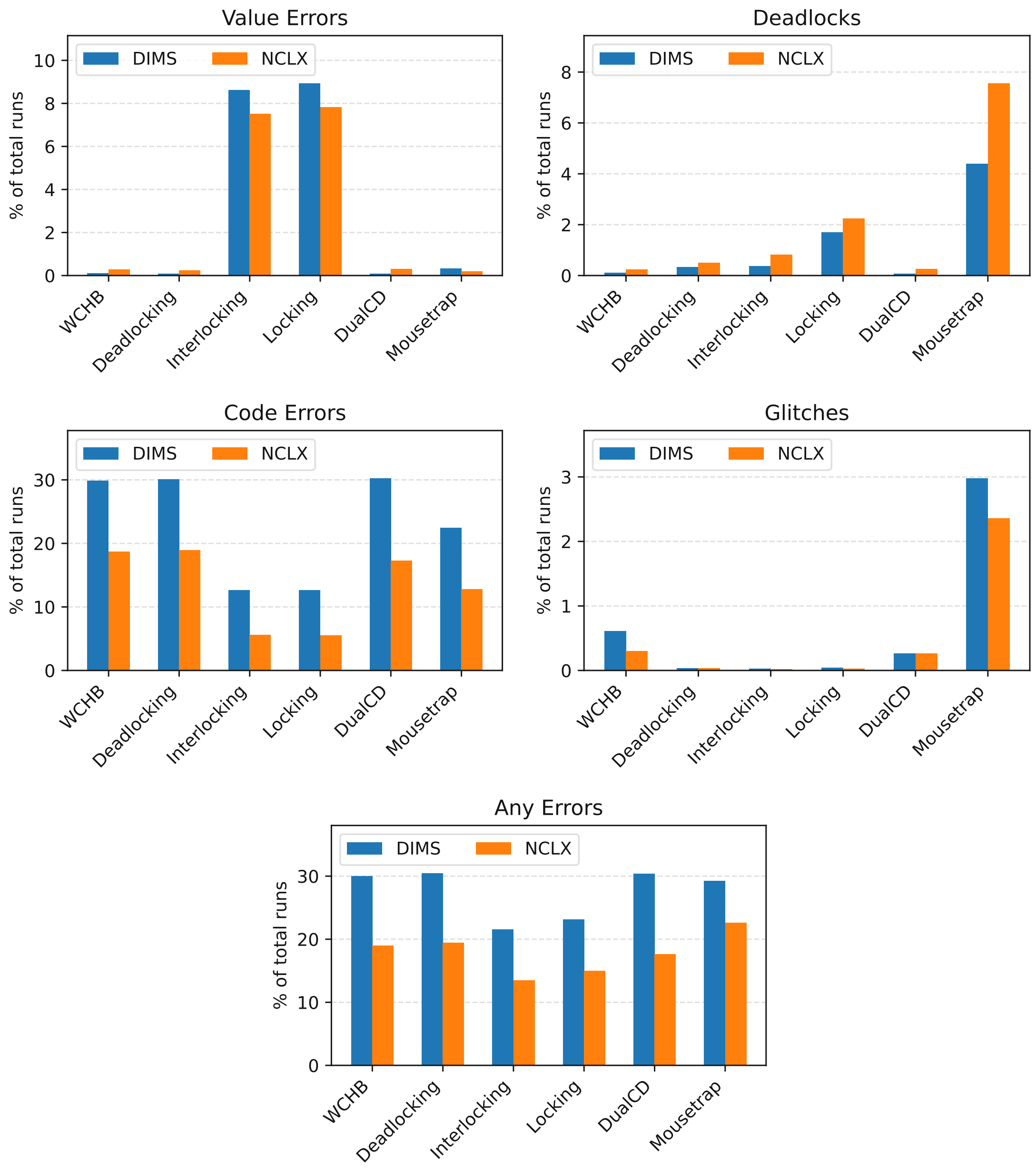


Figure 6.3: 8-bit Multiplier error rates of the different buffer types and logic styles relative to all performed bit flip injections

6.2.1 Effects of the Fault Duration

An interesting property to look at is the effect that the fault duration has on the observed errors. To analyze this, a separate experiment was run on the 4-bit multiplier with the WCHB buffer type both with DIMS and NCLX. As opposed to the other experiments, the total number of fault durations was increased to 100, meaning that fault durations from 5 to 500 ns were tested. Based on these results, the numbers presented in Figure 6.4 were obtained.

It is visible that the fault duration has a significant impact on all error types, with the effect starting to become a lot less notable at around 100 ns for most error types.

One notable exception to this trend are glitches, which hit their highest value much earlier and are the most stable from then on. This is not surprising behavior as glitches occur in transitional states of the protocol. As the individual stages are not that big, a certain fault duration will eventually cover the full time span of the transitions and such a longer fault duration will not cause further glitches.

The other obvious observation to be made is the rather unstable rate of value errors which also seems to hit a plateau but then oscillates and actually shows a small downward trend. This is not as straightforward to explain and without further investigation it is only possible to speculate on the cause of this effect. One possible explanation could be the volatile nature of value errors: Both data rails of a single bit need to switch to the wrong value at least long enough for the data to be captured. With longer fault durations it stands to reason that the timing of the data rail transitions to each other shift just enough that the faults manifest in the form of other errors (e.g., code errors).

A similar behavior is seen on deadlocks which also start oscillating at a certain fault duration. The reasoning could be very similar to the one for value errors but has not been further analyzed. It could be interesting to run these experiments on different buffer types where value errors and deadlocks are generally more common.

6.2.2 Effects of the Output Delay

The last analyzed aspect is the effect of the output delay on error rates. The idea of increasing the output delay is so that the pipeline spends more time in a filled state (i.e., multiple tokens being processed simultaneously) instead of processing data token by token.

It can be seen in Figure 6.5 that the effect strongly depends on the error type as well as the used buffer style. One of the most interesting observations is probably the change in error rates on the interlocking and the locking WCHB compared to the other buffer types. The value error rate drops significantly with the higher output delays while it increases on the other buffer types. For code errors and deadlocks on the other hand, there is a noticeable increase. As the overall error rate remains roughly the same, it seems likely that the code errors and deadlocks replace some of the value errors for higher output delays. The cause of this phenomenon is yet to be determined and would require

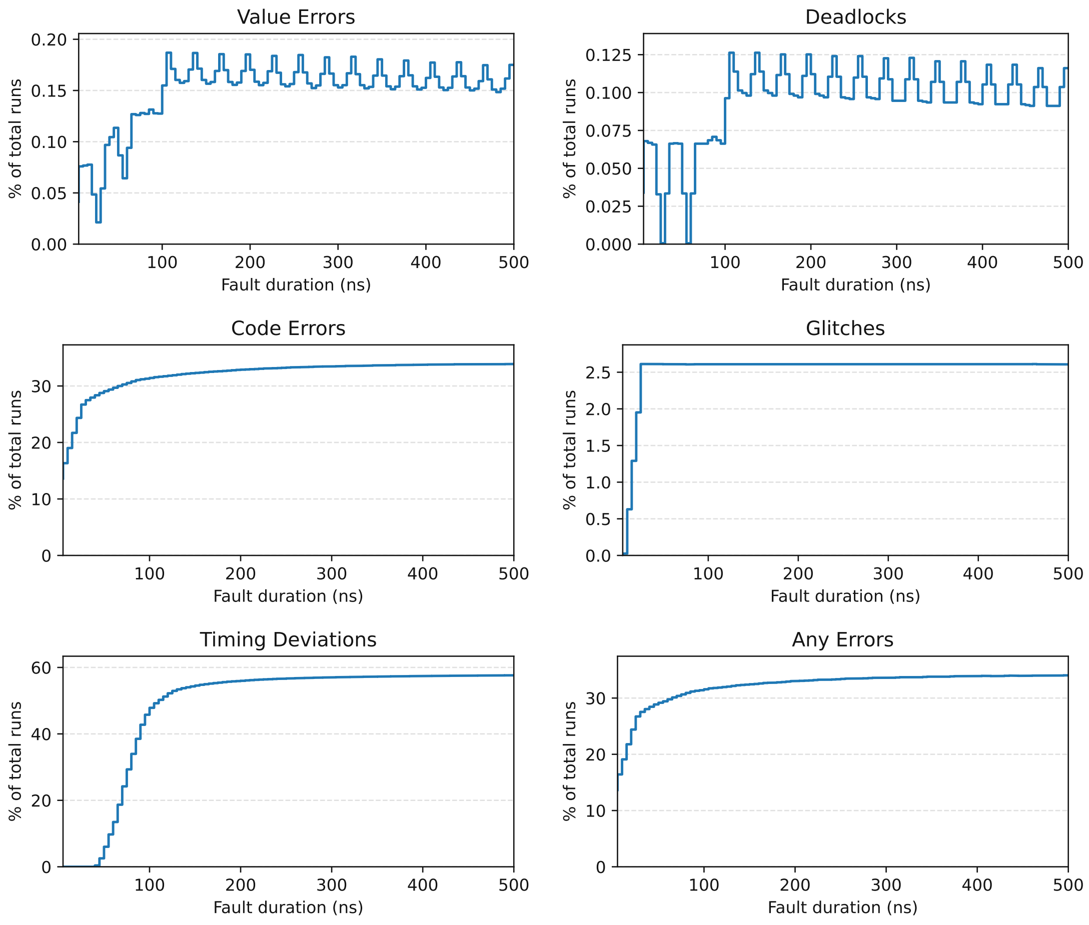


Figure 6.4: 4-bit multiplier error rates (WCHB) in relation to the fault duration

further analysis. It is particularly interesting because this trend was not observed at all in [BHN+21].

The other observation of note is the reduction of glitches on the MDHB with increasing output delays, seeing the error rate fall from 6% to 2%. This indicates that the latches spend less time in transparent states when the pipeline is filling up which also matches the results presented in [BHNS21].

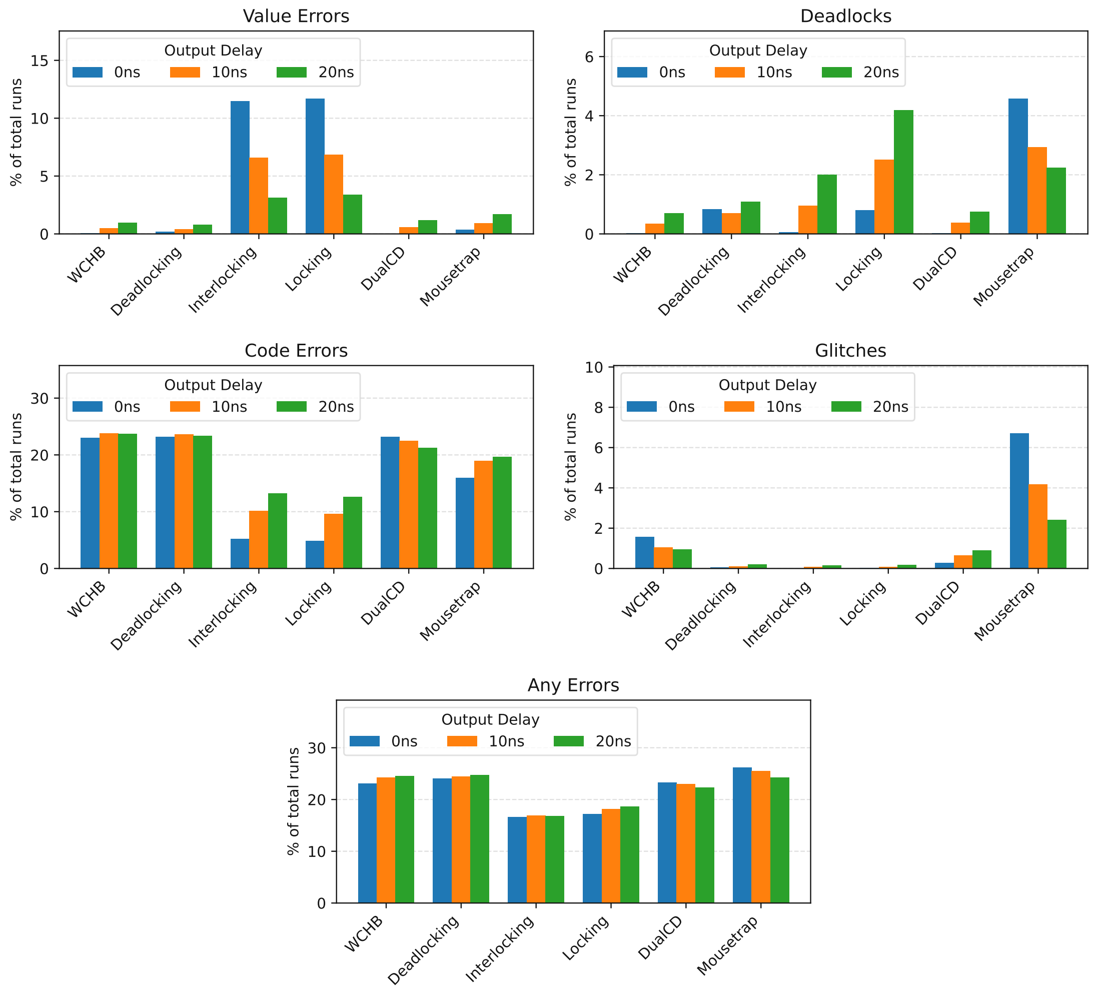


Figure 6.5: Effect of the output delays on the error rates on the 4-bit multiplier

6.3 Carry Ripple Adder

The carry ripple adder differs from the other tested circuits as the basic design is purely combinational and does not use any buffers. Thus, when plugging it into the injection harness, the acknowledgement of the input token is directly connected to the acknowledgement of the output token. As synchronizers are required both on input and output, the overhead introduced by the synchronizers alone would take longer than the actual combinational processing of the adder, skewing the observed fault-injection effects as the circuit would spend most of its time waiting for an acknowledgement. To counter-act this problem, delay gates are added to slow down the circuit as described in Chapter 4.4.3.

The circuit was tested with three different input sets as explained in [MRL04]:

- An exhaustive set, testing all (512) potential input combinations
- A zeros set where $A = B = 0$, where the carry bit does not need to ripple through
- A worst-case set where $A = 0\text{xF}$ and $B = 0$, activating the critical path of the circuit as the fast carry output is not used in any of the full adders

The experiment was executed on a sequence of 544 input tokens, with the fault injections being performed in such a time window that they are performed on the middle 512 input tokens. This is done to ensure that all input combinations on the exhaustive set are tested while avoiding irregular behavior due to being at the beginning or the end of each run.

6.3.1 Input Set Comparison

The first aspect we looked at was the effect of the input set on the different error rates of the circuit. For this experiment, the fault durations were chosen to be between 5 and 230 nanoseconds (starting at 5 ns, using 10 different widths, incrementing by 25 ns each). Ten delay gates were added at the outputs of all gates in the circuit to ensure that the asynchronous circuit was running significantly slower than the injection harness. The extracted results are shown in Table 6.3.

The impact of the input set is clearly visible in how long the golden run takes to finish, with the worst-case input set taking almost 50% longer than the zeros input set and the exhaustive set being almost exactly centered between the two.

The effect can also be seen in the error rates, as the zeros set has the lowest error rate, producing no value errors at all. Interestingly, the exhaustive input set has the highest total error rate, surpassing the worst-case set. This shows that while the combination used for the worst-case set is clearly the slowest, it is not the one that is most sensitive to faults. To understand these differences in error rates it helps to analyze the number of C gates that have a direct impact on the output of one of the full adders. For $A = B = 0$, each full adder has four of ten C gates that would cause an output to switch that is not supposed to. All other C gates do not affect any output signals or only those that are supposed to toggle anyways (i.e., at worst causing a signal to toggle earlier than expected). When $A = 0\text{xF}$ and $B = 0$, that number increases to five C gates, providing an explanation for the increased error rate. The same analysis could be done for the exhaustive set by calculating these values for all 512 input combinations.

6.3.2 Sensitivity Windows of C Gates

As the next step, the intent was to reproduce the analysis presented in [MRL04] in real hardware, more specifically to look at the error rates caused by fault injections on specific

Input Set	Golden Run Time	Value Errors	Any Errors
Zeros	103 us	0.000%	17.543%
Exhaustive	122 us	1.023%	22.142%
Worst-Case	146 us	0.621%	20.836%

Table 6.3: Error rates of the 4-Bit carry ripple adder on different input sets

C gates and comparing them to the amount of time these gates are supposed to be in 1-sensitive states.

To achieve representative results, the same experiment as before was executed but only the shortest possible injection pulse was chosen (i.e., 5 ns), thus each individual fault injection covers exactly one time window without overlaps to other injections. To maximize the granularity of the fault-injection windows, the circuit was generated with 20 delay gates. The numbers shown in [MRL04] and our results are condensed into Table 6.4 to provide a brief overview.

C Gate	Input Set	Time in 1-sensitive state	Error Rate
FA0.M02	Exhaustive	27.75%	12.31%
FA3.M02	Exhaustive	24.97%	12.45%
FA0.M07	Zeros	28.57%	0.00%
FA3.M07	Zeros	0.00%	0.00%
FA0.M09	Worst-Case	14.29%	0.00%
FA3.M09	Worst-Case	54.55%	0.00%

Table 6.4: Adder error rates compared to expected time in 1-sensitive states of C gates

While it was not expected that the error rates would match up exactly with the times the circuit spends in 1-sensitive states, we were hoping to see similar ratios or at least that the rates between the adders would decrease or increase accordingly. It is quite clear that we were unable to achieve the desired results as no errors were observable at all for the M07 and the M09 C gates. This is actually no surprise when looking at the circuit in more detail. Both M07 with the zeros input set and M09 with the worst-case set are gates that should be set in the data phase, meaning that the time they spend in a 1-sensitive to 0 state is just the time before the expected transition to 1. If a fault now causes the transition prematurely, this could at most be noticeable as a timing deviation, but it will not cause an observable error at the output. For the exhaustive input set, a similar effect might occur where the discrepancy in time spent in 1-sensitive states only comes from situations where the 'faulty' transition is bound to happen anyways. This shows that the analysis of 1-sensitive states as performed in [MRL04] does not directly translate to the error rate of circuits.

6.4 Framework Performance

One major motivation behind the framework is achieving a better performance compared to a purely simulation based framework like the one presented in [BHNS21]. To evaluate this, the execution times of the experiments are analyzed and set in relation to the total individual fault injections performed. Assuming a fixed UART baud rate, there are two major factors that affect the number of injections per second: The number of injections performed per synthesis and the observed error rates.

As a new synthesis is required whenever the injection targets change, which takes roughly a constant amount of time, it is evident that the injection rate can be improved by increasing the number of injections per synthesis, most notably by increasing the following parameters:

- Number of error gates per synthesis run
- Number of pulse start times
- Number of pulse widths
- Number of input and output delay combinations

The overall error rate is the second performance factor due to the UART interface being an inherent bottleneck for the framework, meaning that once the result FIFO is full, an entry needs to be transmitted to the workstation before another injection can be started, slowing down the actual execution. Thus, a circuit that exhibits low error rates should generally have a better injection rate.

The values shown in Table 6.5 were obtained when running the experiments as presented earlier in this chapter. It can be seen that the overall injection rate never fell below 4,000 injections per second. It is also clearly visible that the number of injection gates used massively increases the injection rate.

The second entry for the multiplier circuit comes from the experiments run for determining the fault duration effect in Section 6.2.1. Here the main reason for the lower injection rate is the high number of timing deviations, being higher than 50% for almost 80% of all runs, leading to huge overhead in data that needs to be transmitted via UART.

To really show the effect of the UART bottleneck, the fault duration experiment was re-run such that the faults would not cause any errors, i.e. for each execution only the golden run result would need to be transmitted (this was done by setting the initial pulse start higher than the golden run takes to execute). The effect is very clearly visible, as the injection rate more than triples. This even slightly understates the effect of the slow UART interface because just the synthesis takes up around 70 seconds of each run. The actual experiment execution takes roughly 190 seconds with and 25 seconds without any errors being reported.

Circuit	Gates	Starts	Widths	Injections	Duration	Injections/s
LFSR	32	440	10	233 million	16 hours	4,046
Multiplier	32	447	10	144 million	6.3 hours	6,330
Multiplier	32	385	100	35 million	2.2 hours	4,378
Multiplier (no err.)	32	385	100	35 million	0.8 hours	18,615
Multiplier (8bit)	128	488	10	646 million	20.3 hours	8,818

Table 6.5: Injection rates of the different experiments that have been run

A direct comparison of the performance to the simulation framework introduced in [BHNS21] is not that easy as such numbers were not directly presented. Instead, we look at the shown simulation progress over a full month. As there were initial issues which were eventually resolved, it is not fully representative to look at the full month. As such, only the period from April 13th, 2021 to April 17th, 2021 will be looked at, which seems to be when the framework was working at its best. During that time period, the simulation count grew from roughly 0.44 to 0.74 billion, meaning that 300 million simulations were performed in a span of 4 days. These numbers lead to an estimated rate of 868 fault injections per second.

At first glance, these numbers may seem underwhelming as the FPGA framework in some cases only performs about five times faster while the fault injection is a lot less precise. It needs to be considered that the simulation framework employs a very high degree of automation, spreading the simulation over 40 computers running in parallel and storing results in a common SQL database. The results presented here, on the other hand, were all performed on a single board, without any parallelization between the individual runs. Some options for potentially increasing the framework performance are explored in Chapter 7.2.

CHAPTER 7

Conclusion and Outlook

The final chapter provides an overview of the research that has been done in the course of this thesis and also gives some outlook on potential improvements that could be implemented in the future.

7.1 Conclusion

With the developed fault-injection harness and accompanying Python library, it was possible to execute billions of individual fault-injection experiments, taking just a few days of execution time. Especially the Python library turned out to be a big factor, automating the entire process of experiment execution:

- Iteration over different circuits and parameters
- Circuit augmentation
- Synthesis of injection harness and download to FPGA
- Experiment execution
- Result retrieval and processing

The results themselves are stored in simple CSV files, making them easily accessible without the need for specific software. In addition, the framework provides functionality for basic filtering of the results and generating summaries of error rates for easy interpretation of generated results.

In the process of evaluating results for this thesis, it became clear that the handling of such big data sets can get cumbersome especially when attempting to filter for very specific cases. Thus, it is generally advisable to keep the experiment configuration in clearly defined scopes, depending on the observations that the user is trying to make.

The results presented in this thesis are not aimed at giving answers about which buffer or logic style to use, but rather to prove that the framework obtains realistic results compared to previous publications and to show some of the specific capabilities that the framework itself has to offer, e.g.:

- Injecting faults on circuits without input
- Precisely targeting certain signals for fault injection
- Result processing for specific victims

While the high degree of automation allowed the execution of hundreds of millions of injections without user intervention, the framework can also be used in a more iterative process. By generating results within minutes, it has the potential to be an effective tool when trying to find the right fault mitigation strategy for a QDI circuit.

7.2 Outlook

During development of the framework, numerous ideas for improvements and further work came up, which exceeded the scope of this thesis so they have not been further explored yet:

- **Supporting a wider range of QDI circuits**
  As outlined in Chapter 4.1, there are specific limitations that the tested circuit needs to fulfill to be compatible to our framework. Some limitations that could be interesting to widen is allowing multiple input/output channels or supporting different QDI protocols, e.g., a 2-phase DR protocol.
- **Improving UUT performance**
  One issue with the implementation stems from the communication between the UUT and the injection harness as the input and output tokens cross the clock boundary, requiring synchronizers. These synchronizers add delays to the acknowledgement of input and output tokens, potentially being a bottleneck for the actual execution of the QDI circuit. A potential solution for this could be the use of dual-port FIFOs on both sides of the UUT for communicating the tokens, providing an asynchronous interface on one side and a synchronous interface on the other. This would allow reading and writing the tokens without the need for synchronizers, keeping the entire execution sequence in the asynchronous domain.
- **Changing the host interface**
  As described earlier, the UART interface is the most obvious bottleneck when it

comes to experiment execution, as the harness regularly needs to wait for the result FIFO to empty before starting the next injection. This problem could be avoided by, e.g., using a USB interface, improving the result transmission rate. This change could also lead to an on-demand interface for the results instead of constantly pushing them. That can be an important factor when it comes to processing the results on the host, as the current implementation needs to ensure that no bytes are missed on the UART interface, effectively waiting for all Bytes to be transmitted before starting the processing of any results.

- **Use a database for results storage**
  While the usage of CSV files allows for very easy information access and storage, they are not very efficient when it comes to receiving detailed results. Setting up a proper SQL database could improve performance of results processing and analysis.

- **Parallelization**
  Another obvious way of improving the performance of experiment execution is adding parallelization capabilities. The potential scope of this option is very wide, ranging from the possibility of running the synthesis for the next experiment while the current one is still executing to having multiple workstations that coordinate through a central database similar to what is presented in [BHNS21].

List of Figures

List of Tables

Acronyms

ASIC	Application-Specific Integrated Circuit. 17
BD	Bundled Data. 6, 8, 69
CD	Completion Detector. 7, 31, 46, 47
CSV	Comma Separated Values. 36, 37, 65, 67
DFG	Data Flow Graph. 32
DI	Delay Insensitive. 7, 8
DIMS	Delay-Insensitive Minterm Synthesis. 9–11, 32, 33, 38, 41, 42, 44, 49, 50, 52, 53, 57
DR	Dual-Rail. 7–10, 12, 13, 15, 23, 31, 33, 42, 43, 45–48, 50, 66, 69, 70
EDA	Electronic Design Automation. 33
FF	Flip-Flop. 3, 4
FIFO	First In First Out. 19, 26, 31, 62, 66, 67
FPGA	Field Programmable Gate Array. 2, 13, 15, 17, 18, 23, 24, 27, 33, 35, 36, 39, 40, 63, 65
JSON	JavaScript Object Notation. 25, 36
LFSR	Linear-Feedback Shift Register. ix, 41, 42, 49–54, 63, 69–71
LUT	Lookup Table. 33, 40
MDHB	MOUSETRAP-Style D-Latch Half Buffer. 47, 48, 50, 51, 58, 70
NCL	Null Convention Logic. 10, 11, 33

NCLX Null Convention Logic with Explicit Completion. 10, 11, 33, 38, 41, 42, 44, 49, 50, 52, 53, 57

PRS Production Rule Set. 32–36, 38, 40

QDI Quasi Delay Insensitive. 1, 8, 9, 12, 13, 15, 16, 19, 20, 23–25, 31–33, 41, 47, 66

RAM Random Access Memory. 18, 31

SDFS Static Data-Flow Structure. 11, 12, 42, 44, 69

SET Single Event Transient. 1, 2, 12, 50, 53, 54

SEU Single Event Upset. 1, 12, 50

SI Speed Independent. 8

SQL Structured Query Language. 21, 63, 67

ST Self-Timed. 8

UART Universal Asynchronous Receiver Transmitter. 18, 24–27, 30, 40, 62, 66, 67, 69

USB Universal Serial Bus. 18

UUT Unit Under Test. 17–19, 23–25, 27, 30–32, 35, 39, 66

VHDL Very High Speed Integrated Circuit Hardware Description Language. 18, 23, 24, 32, 33, 35, 36

WCHB Weak-Conditioned Half Buffer. 9, 21, 38, 44–47, 50, 54, 57, 58, 69, 70

Bibliography

[Bau05] Robert C Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3):305–316, 2005.

[Beh21] Patrick Behal. *Quantitative Comparison of the Sensitivity of Delay-Insensitive Design Templates to Transient Faults*. PhD thesis, Wien, 2021.

[BHN$^+$21] Patrick Behal, Florian Huemer, Robert Najvirt, Andreas Steininger, and Zaheer Tabassam. Towards explaining the fault sensitivity of different qdi pipeline styles. In *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 25–33. IEEE, 2021.

[BHNS21] Patrick Behal, Florian Huemer, Robert Najvirt, and Andreas Steininger. An automated setup for large-scale simulation-based fault-injection experiments on asynchronous digital circuits. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 541–548. IEEE, 2021.

[BS09] William John Bainbridge and Sean James Salisbury. Glitch sensitivity and defense of quasi delay-insensitive network-on-chip links. In *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 35–44. IEEE, 2009.

[EGRM20] Mohammad Eslami, Behnam Ghavami, Mohsen Raji, and Ali Mahani. A survey on fault injection methods of digital integrated circuits. *Integration*, 71:154–163, 2020.

[FB96] Karl M Fant and Scott A Brandt. Null convention logic/sup tm: a complete and consistent logic for asynchronous digital circuit synthesis. In *Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP'96*, pages 261–273. IEEE, 1996.

[FCMG13] Veronique Ferlet-Cavrois, Lloyd W Massengill, and Pascale Gouker. Single event transients in digital cmos—a review. *IEEE Transactions on Nuclear Science*, 60(3):1767–1790, 2013.

[Gin11] Ran Ginosar. Metastability and synchronizers: A tutorial. *IEEE Design & Test of Computers*, 28(5):23–35, 2011.

[HNS20] Florian Huemer, Robert Najvirt, and Andreas Steininger. Identification and confinement of fault sensitivity windows in qdi logic. In *2020 Austrochip Workshop on Microelectronics (Austrochip)*, pages 29–36. IEEE, 2020.

[Hue22] Florian Ferdinand Huemer. *Contributions to efficiency and robustness of quasi delay-insensitive circuits*. PhD thesis, Technische Universität Wien, 2022.

[KAN11] Jagrit Kathuria, M Ayoubkhan, and Arti Noor. A review of clock gating techniques. *MIT International Journal of Electronics and Communication Engineering*, 1(2):106–114, 2011.

[KL02] Alex Kondratyev and Kelvin Lwin. Design of asynchronous circuits by synchronous cad tools. In *Proceedings of the 39th annual Design Automation Conference*, pages 411–414, 2002.

[Mar90] Alain J Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, pages 302–311. Springer, 1990.

[MN06] Alain J Martin and Mika Nystrom. Asynchronous techniques for system-on-chip design. *Proceedings of the IEEE*, 94(6):1089–1120, 2006.

[MRL04] Yannick Monnet, Marc Renaudin, and Régis Leveugle. Asynchronous circuits sensitivity to fault injection. In *Proceedings. 10th IEEE International On-Line Testing Symposium*, pages 121–126. IEEE, 2004.

[Mul59] David Muller. A theory of asynchronous circuits. In *Proc. Int. Symp. on Theory of Switching, 1959*, volume 29, pages 204–243, 1959.

[ORG92] Joakim Ohlsson, Marcus Rimen, and Ulf Gunneflo. A study of the effects of transient fault injection into a 32-bit risc with built-in watchdog. In *FTCS*, volume 22, pages 316–325, 1992.

[PHRB11] Ilia Polian, John P Hayes, Sudhakar M Reddy, and Bernd Becker. Modeling and mitigating transient errors in logic circuits. *IEEE Transactions on Dependable and Secure Computing*, 8(4):537–547, 2011.

[RS08] Babak Rahbaran and Andreas Steininger. Is asynchronous logic more robust than synchronous logic? *IEEE Transactions on dependable and secure computing*, 6(4):282–294, 2008.

[RSH04] Babak Rahbaran, Andreas Steininger, and Thomas Handl. Built-in fault injection in hardware-the fidyco example. In *Proceedings. DELTA 2004. Second IEEE International Workshop on Electronic Design, Test and Applications*, pages 327–332. IEEE, 2004.

[SN07] Montek Singh and Steven M Nowick. Mousetrap: High-speed transition-signaling asynchronous pipelines. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(6):684–698, 2007.

[Spa20] Jens Sparsø. *Introduction to Asynchronous Circuit Design.* DTU Compute, Technical University of Denmark, 2020.

[SS93] Jens Sparsø and Jørgen Staunstrup. Delay-insensitive multi-ring structures. *Integration*, 15(3):313–340, 1993.

[SS11] Jitesh Shinde and SS Salankar. Clock gating—a power optimizing technique for vlsi circuits. In *2011 annual IEEE India conference*, pages 1–4. IEEE, 2011.

[WQR+04] Nicholas J Wang, Justin Quek, Todd M Rafacz, et al. Characterizing the effects of transient faults on a high-performance processor pipeline. In *International Conference on Dependable Systems and Networks, 2004*, pages 61–61. IEEE Computer Society, 2004.