UNIVERSIDAD POLITÉCNICA DE VALENCIA

DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

SPEEDING-UP MODEL-BASED FAULT INJECTION OF DEEP-SUBMICRON CMOS FAULT MODELS THROUGH DYNAMIC AND PARTIALLY RECONFIGURABLE FPGAS

Mr. David de Andrés Martínez

Thesis directors
Prof. Pedro Joaquín Gil Vicente
Dr. Juan Carlos Ruiz García

Valencia, 2007

Resumen

Actualmente, las tecnologías CMOS submicrónicas son básicas para el desarrollo de los modernos sistemas basados en computadores, cuyo uso simplifica enormemente nuestra vida diaria en una gran variedad de entornos, como el gobierno, comercio y banca electrónicos, y el transporte terrestre y aeroespacial. La continua reducción del tamaño de los transistores ha permitido reducir su consumo y aumentar su frecuencia de funcionamiento, obteniendo por ello un mayor rendimiento global. Sin embargo, estas mismas características que mejoran el rendimiento del sistema, afectan negativamente a su confiabilidad. El uso de transistores de tamaño reducido, bajo consumo y alta velocidad, está incrementando la diversidad de fallos que pueden afectar al sistema y su probabilidad de aparición. Por lo tanto, existe un gran interés en desarrollar nuevas y eficientes técnicas para evaluar la confiabilidad, en presencia de fallos, de sistemas fabricados mediante tecnologías submicrónicas.

Este problema puede abordarse por medio de la introducción deliberada de fallos en el sistema, técnica conocida como inyección de fallos. En este contexto, la inyección basada en modelos resulta muy interesante, ya que permite evaluar la confiabilidad del sistema en las primeras etapas de su ciclo de desarrollo, reduciendo por tanto el coste asociado a la corrección de errores. Sin embargo, el tiempo de simulación de modelos grandes y complejos imposibilita su aplicación en un gran número de ocasiones.

Esta tesis se centra en el uso de dispositivos lógicos programables de tipo FPGA (*Field-Programmable Gate Arrays*) para acelerar los experimentos de inyección de fallos basados en simulación por medio de su implementación en hardware reconfigurable. Para ello, se extiende la investigación existente en inyección de fallos basada en FPGA en dos direcciones distintas: i) se realiza un estudio de las tecnologías submicrónicas existentes para obtener un conjunto representativo de modelos de fallos transitorios y permanentes que deben ser tenidos en cuenta, y se analiza hasta que punto estos modelos pueden emularse por medio de FPGA, y ii) para cada modelo de fallo considerado, se describen diversos procedimientos alternativos para realizar su emulación, evaluando la aceleración obtenida en cada caso.

FADES (*F*PGA-based Framework for the *A*ssessment of the *D*ependability of *E*mbedded *S*ystems) es el prototipo desarrollado para ilustrar la viabilidad de esta metodología. Los experimentos realizados por medio de FADES se han centrado en comprobar la validez de los resultados obtenidos y mostrar la aceleración alcanzable por comparación con una herramienta de inyección de fallos reconocida. Finalmente, se deducen las principales ventajas e inconvenientes de la aproximación propuesta, así como su utilidad para la evaluación de la confiabilidad de sistemas empotrados.

Resum

En l'actualitat, les tecnologies CMOS submicròniques són bàsiques pel desenvolupament dels moderns sistemes basats en computadors, els quals simplifiquen enormement la nostra vida diària en una gran varietat d'entorns, com el govern, comerç i banca electrònics, i el transport terrestre i aeroespacial. La contínua reducció del tamany dels transistors ha permès reduir el seu consum i augmentar la seua freqüència de funcionament, obtenint un millor rendiment global. No obstant açò, aquestes característiques que milloren el rendiment del sistema, afecten negativament a la seua confiabilitat. L'ús de transistors de tamany reduït, baix consum i alta velocitat, incrementa la diversitat de fallades que poden afectar al sistema i la seua probabilitat d'aparició. Per tant, existeix un gran interés en desenvolupar noves i eficients tècniques per avaluar la confiabilitat, en presència de fallades, de sistemes fabricats mitjançant tecnologies submicròniques.

Aquest problema pot abordar-se mitjançant la introducció deliberada de fallades en el sistema, tècnica coneguda com injecció de fallades. En aquest context, la injecció basada en models és molt interesant, ja que permet avaluar la confiabilitat del sistema en les primeres etapes del seu cicle de desenvolupament, reduint per tant el cost associat a la correcció d'errors. Malgrat açò, el temps de simulació de models grans i complexos impossibilita la seua aplicació en un gran nombre d'ocasions.

Aquesta tesi es centra en l'ús de dispositius lógics programables de tipus FPGA (*Field-Programmable Gate Arrays*) per tal d'accelerar els experiments d'injecció de fallades basats en simulació, mitjantçant la seua implementació en hardware reconfigurable. La investigació existent en injecció de fallades basada en FPGA s'ha estés en dues direccions diferents: i) s'ha realitzat un estudi de les tecnologies submicròniques existents per tal d'obtindre un conjunt representatiu de models de fallades transitèries i permanents que cal tenir en compte, i s'ha analitzat fins a quin punt aquests models poden emular-se mitjançant una FPGA, i ii) per cada model de fallada considerat, s'han descrit diversos procediments alternatius per realitzar la seua emulació, avaluant l'acceleració obtinguda en cada cas.

FADES (*F*PGA-based Framework for the *A*ssessment of the *D*ependability of *E*mbedded *S*ystems) és el prototip desenvolupat per il·lustrar la viabilitat d'aquesta metodologia. Els experiments realitzats mitjançant FADES s'han centrat en comprobar la validesa dels resultats obtinguts i mostrar l'acceleració assolible per comparació amb una eina d'injecció de fallades reconeguda. Finalment, s'han deduit els principals avantatges i inconvenients de l'aproximació proposta, i la seua utilitat per tal d'avaluar la confiabilitat de sistemes empotrats.

Abstract

Nowadays, deep-submicron CMOS technologies are basic for the development of modern computer-based systems, whose use simplify our everyday life in a variety of domains such as e-government, electronic commerce and banking and terrestrial and aerospace transportation. The steady reduction of transistors size allows for less power consumption and higher frequency rates, leading altogether to greater performance. These same features that improve VLSI systems performance, negatively affect their dependability. Low-power, high-speed, reduced-size transistors are highly increasing the likelihood of occurrence and the diversity of faults affecting these systems. Therefore, there exists a great interest in developing new and efficient techniques to assess the dependability of deep-submicron manufactured systems in the presence of faults.

Fault injection, which consists in the deliberate introduction of faults into a system, is a well-known approach for coping with this problem. In this context, model-based fault injection has the interest of enabling the dependability assessment of the system in the early stages of its development cycle, thus reducing the costs of fixing any error. However, the long time required to simulate large and complex models make it impractical in many cases.

This thesis focuses on the use of Field-Programmable Gate Arrays (FPGAs) to accelerate model-based fault injection experiments by implementing the model of the system under study on reconfigurable hardware. It improves existing research on FPGA-based fault injection in two different directions: i) it studies existing semiconductor technologies to determine a representative set of transient and permanent fault models to cope with, and analyses to what extent such models, can be emulated by means of FPGAs, and ii) it determines, for each considered fault model, alternative procedures to carry out its emulation, evaluating the resulting speed-up.

FADES (*F*PGA-based Framework for the *A*ssessment of the *D*ependability of *E*mbedded *S*ystems) is the prototype that has been developed in order to illustrate the feasibility of the approach. The experimentation deployed using FADES is mainly focused on validating the correctness of the results provided by the proposed prototype and showing the attainable speed-up of the solution when compared with a state of the art simulation-based fault injection tool. Finally, a discussion reflecting the main advantages and drawbacks of the proposed approach and its usefulness for the dependability assessment of embedded systems is provided.

Endure. In enduring, I grow stronger.
Teachings of Zerthimon, Planescape Torment.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ALU Arithmetic-Logic Unit

API Application Programming Interface

ASIC Application Specific Integrated Circuit

BIST Built-In Self Test

CB Configurable Block

CISC Complex Instruction Set Computer

CMOS Complementary Metal-Oxide Semiconductor

COTS Components-Off-the-Shelf

DSP Digital Signal Processing

EEMBC Embedded Microprocessor Benchmark Consortium

FADES FPGA-based framework for the Assessment of the Dependability of Embedded Systems

FF Flip-Flop

FPGA Field Programmable Gate Array

GSR Global Set Reset

GUI Graphical User Interface

HDL Hardware Description Language

HWIFI Hardware Implemented Fault Injection

ISE Integrated Software Environment

LSR Local Set Reset

LUT Look-Up Table

MAC Multiply-and-Accumulate

PM Programmable Matrix

PT Pass Transistor

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

SET Single Event Transient

SEU Single Event Upset

SoC System on a Chip

SRAM Static Random Access Memory

SWIFI Software Implemented Fault Injection

VHDL Very High Speed Integrated Circuits Hardware Description Language

VLSI Very Large Scale of Integration

XHWIF Xilinx standard Hardware Interface

Chapter 1

Introduction

Continuous advances in semiconductor technologies have drawn computer-based systems into everyday life and, nowadays, these systems are widely used in almost any application domain. The design of most computer-based systems has been traditionally focused on increasing their performance to satisfy the market needs. Besides increasing the functionality/area ratio, the steady reduction of transistors size has also allowed for less power consumption and higher frequency rates, leading to greater performance for a given function.

However, there exist a large number of computer-based systems, such as airplanes, space probes, or nuclear power plants, that cannot be designed under this same principle. A failure in these systems is totally unacceptable, since people depend on them, often for their livelihood, sometimes for their lives. Hence, the design of critical systems must be focused on improving its dependability rather than its performance.

While critical systems raise obvious dependability concerns, due to the widespread use of computer-based systems, unexpected or premature failures in even non-critical applications, such as video game consoles and portable video players, can greatly damage the reputation of the manufacturer and reduce the acceptability of new products. So, nowadays, achieving a certain degree of dependability is also a must for non-critical application domains like consumer electronics.

Although improving the performance of computer-based systems, low-power, high-speed, reduced-size transistors are also negatively impacting their dependability. On one hand, the likelihood of faults occurrence is greatly increasing in deep submicron manufactured systems. On the other hand, new and more complex fault models are resulting from the fault causes and mechanisms related to these emerging technologies.

Therefore it is not only necessary to study the functional behaviour of semiconductor manufactured systems, but also their behaviour in the presence of representative faults as ultimate cause of system's failure, such as *bit-flips*, *pulses*, *stuck-ats*, *stuck-opens*, *indeterminations*, *delays*, *shorts*, *open-lines*, and *bridgings*.

Fault injection, which consists in the deliberate introduction of faults into a system, has been largely recognised as a suitable approach to assess the robustness and validate the dependability of computer-based systems.

Prototype-based fault injection techniques require a prototype to introduce faults into the system. These techniques can only be applied in the last stages of the development cycle, thus increasing the costs of fixing any error. Usually, they are not very flexible and the set of available fault models is very restricted.

Model-based fault injection techniques introduce the faults into a model of the system. This approach can be applied along the development cycle, thus reducing the costs of fixing any error in the design. It is also very flexible, allowing for the injection of a wide variety of faults. However, the simulation of the model can take so long that it could result impracticable in many cases.

As the steady reduction of transistors size is leading to larger and more complex systems, new fault injection techniques joining together the speed of prototypes and the flexibility of models should be developed.

Field-Programmable Gate Arrays (FPGAs), which have been used for prototyping over the years, offer a viable solution to solve that particular problem. One of the main application domains of FPGA-based computing is determining whether the current implementation of a logic circuit fulfills the requirements specified at the design stage (logic validation).

Previous existing logic validation techniques included *hardware-prototyping* and *software simulation*, which present the same benefits and drawbacks as prototype-based and model-based fault injection techniques, respectively.

Logic emulation consists in the use of FPGAs to implement the model of the system. It is emulated much faster than in the case of software simulation, and it is also easier, cheaper and faster than building its prototype. All these reasons make FPGAs well-suited devices for the logic validation of circuits.

It is to note the resemblance in the methodologies used in fault injection and logic validation techniques. Hence, why not use FPGAs to implement the model of the system and thus accelerate the execution of model-based fault injection experiments? It should also be easier and faster than prototype-based fault injection techniques. That technique, which presents the same basic characteristics as logic emulation for logic validation, is named *fault emulation*.

At present, two different methodologies for fault emulation have evolved from the existing approaches for the design of reconfigurable applications: *Compile-* and *Run-Time Reconfiguration*.

The former relies on the model instrumentation in order to inject faults into the system and the latter dynamically reallocates hardware on the programmable device.

Both techniques present complementary advantages and drawbacks, but they both have been mainly focused on injecting the well-known *bit-flips* and *stuck-ats*. The emulation of the rest of fault models considered representative of deep-submicron manufactured systems have not been addressed yet.

Taking all this aspects into account, the main goal of this thesis is **analysing to what extent fault models, derived from new semiconductor technologies, can be emulated by means of FPGAs to minimise the time required to simulate the behaviour of the system in the presence of that faults.**

Although FPGAs could be used with fault injection purposes, it is necessary to determine if they are flexible enough to deal with the whole set of hardware fault models considered representative of deep submicron manufactured systems. As the system's model will be implemented on reconfigurable hardware, that will accelerate its execution.

So, fault emulation methodologies will be thoroughly studied to determine which is the most suitable methodology for the design of reconfigurable applications that could be applied for emulating the whole set of considered faults.

Then, a detailed study will report, for each one of these faults, i) the elements of the system's model that can be targeted by the fault, ii) which reconfigurable components of the FPGA these model's elements map to, and iii) how these FPGA's components should be reconfigured to emulate the behaviour of the system in the presence of that fault following the selected methodology. Novel approaches for those fault models that were already addressed by fault emulation, and different approaches for the injection of the rest of the faults will also be detailed.

An account of the time required to reconfigure the FPGA when following each of the proposed approaches is also mandatory. This account will enable the comparison between the different approaches for injecting a particular fault and the definition of guidelines for selecting which is the most suitable one for accelerating the execution of the system's model.

The achievement of this first goal poses the necessity of **providing an implementation of a tool that automates the proposed methodology**

for assessing the dependability of computer-based systems in the presence of representative faults.

Then, a second goal is developing a tool that implements the considered methodology and all the proposed approaches for injecting the whole set of hardware fault models by means of FPGAs. That tool should present a very simple graphical user interface to enable its use by non-skilled users.

The first prototype of this tool will be named FADES (**F**ramework for the **A**ssessment of the **D**ependability of **E**mbedded **S**ystems), and it will be written in Java to assure its portability across platforms.

A case study will show the feasibility of using this tool to assess the dependability of computer-based systems. The main advantages and drawbacks of the current implementation of FADES, as well as the more relevant technical problems encountered, will be also reported.

This thesis achieves the previously presented goals and describes all the studies and work performed by following the ensuing structure:

Chapter 2 discusses the set of fault models considered representative of deep submicron technologies, and that will be used throughout this work.

The aim of Chapter 3 is to analyse existing FPGA-based fault injection methodologies to identify which is the most suitable technique coping with our objectives. A brief overview of the FPGAs general architecture and common design flow is first introduced to better understand the advantages and drawbacks of the analysed methodologies. Finally, a generic FPGA architecture is defined to support the definition of new approaches for the injection of significative faults according to the selected methodology.

Chapter 4 defines a general methodology for the emulation of transient and permanent faults. Different approaches are proposed for emulating each particular transient (*bit-flip*, *pulse*, *indetermination*, and *delay*) and permanent (*stuck-at*, *stuck-open*, *indetermination*, *delay*, *short*, *open-line*, and *bridging*) fault, according to the FPGA generic architecture. The applicability and temporal costs of each approach is also reported.

Then, Chapter 5 presents the architecture and control flow of a fault injection tool named FADES, which implements the proposed methodology and the different approaches defined for injecting the whole set of studied fault models.

The first prototype of FADES is experimentally validated in Chapter 6, where it is effectively used to evaluate the dependability of different microprocessor-based systems. Results from this study demonstrate the feasibility of the approach and show the benefits and drawbacks of its implementation.

Finally, Chapter 7 summarises the main conclusions of this thesis and identifies some open challenges for further research.

Chapter 2

Deep-Submicron CMOS Fault Models

Not only the rate of faults occurrence in deep submicron manufactured systems is increasing, but also new and more complex fault models are required to accurately assess the dependability of these kind of systems.

This Chapter presents a wide set of hardware fault models considered representative of new semiconductor technologies that will be used along this thesis.

2.1 Introduction

The same features of deep submicron technologies that contribute to improve the performance of computer-based systems, negatively affect their dependability.

Low-power, high-speed, reduced-size transistors are highly increasing the likelihood of occurrence of transient faults in these systems [1]. Even though improvements in the manufacturing process are somehow limiting the probability of occurrence of permanent faults due to defects, the increasing number of intermittent faults finally manifesting as permanent ones, and the acceleration of the aging process, are also leading to an important increase of the occurrence rate of permanent faults.

As technology evolves toward the nanoelectronics realm, the manufacturing process cannot be so tightly controlled, which will lead to a larger defect rate, and the even smaller device's size will keep increasing the likelihood of occurrence of transient faults.

Hence, studying the behaviour of systems in the presence of representative hardware faults is very important nowadays and will become a must in the near future.

The activation of a *fault*, due to development, physical or interaction reasons, causes an *error*. Any deviation of the delivered service from the correct one, due to error propagation, is called a *failure* [2].

Hence, an error occurring in a component A may propagate to another component B, receiving service from A, when the error reaches the service interface of A. Component A fails to deliver its service correctly and, this failure of A, appears as a fault to B, that propagates the error through its interface. These mechanisms are depicted in Figure 2.1, which shows the causality relationship among faults, errors, and failures.

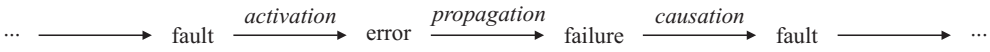

Figure 2.1: *Fundamental chain of dependability threats.*

Therefore it is not only necessary to study the functional behaviour of deep submicron manufactured systems, but also their behaviour in the presence of representative faults as ultimate cause of system's failure.

Fault injection, which consists in the deliberate introduction of faults into a system, has been largely recognised as a suitable approach for dependability assessment.

Nevertheless, the characterisation of the behaviour of a computer-based system in presence of faults, does not require that the injected faults be "close" to the target (reference) faults [3]. As similar errors can be induced by different types of faults, it is enough that these faults induce similar behaviours. For instance, a modification of the contents of a register or memory cell may be provoked by a heavy-ion or as the result of an error provoked by a software fault. What really matters is not to establish an equivalence in the fault domain, but rather in the error domain.

Computer-based architectures are usually represented as consisting of different abstraction layers, from the *application layer* (at the top) to the *hardware layer* (at the bottom). The hardware layer can be further divided into three different levels: the Register Transfer (RT), logic, and physical device levels (top to bottom).

The RT level defines the behaviour of synchronous digital circuits in terms of the flow of information between hardware registers. The logic level, also known as gate level, specifies the behaviour of the circuit by means of logic equations or logic gates and their interconnections. The physical device or transistor level defines the circuit according to the basic building blocks of

the current technology, CMOS (Complementary Metal-Oxide Semiconductor) transistors nowadays, and a netlist with their interconnection.

The main aim of this work is to study how FPGAs can be used for emulating the occurrence of hardware faults considered representative of deep submicron technologies. Hence, as a previous step to this study, it is necessary to define accurate fault models for all the considered hardware faults.

The best option to increase the representativeness of the results of this study should be to determine how these faults manifest at the physical device level, since it is the description level closest to the underlying technology. However, although some work exists in this domain, FPGAs are not suitable devices to implement systems described in a physical device level.

Hence, we will consider how these faults manifest at the logic and RT levels instead, which better match the internal structure of FPGAs (as it will be seen in Chapter 3). These levels are also part of the hardware layer and will keep the representativeness of the study.

A representativeness study [4] from the Dependability Benchmarking project[1] analysed the physical causes and mechanisms of hardware faults in current VLSI systems to determine how they manifest at the logic and RT levels. This comprehensive study determines the set of fault models that are going to be considered along this work.

Section 2.2 discusses those hardware fault models related to transient faults, whereas Section 2.3 describes hardware fault models regarding permanent and intermittent faults. The description and nature of all these hardware fault models are summarised in Section 2.4.

2.2 Transient Faults

Transient faults appear during the normal operation of the circuit for a short period of time after which they disappear again. They usually result from the interference or interaction of the circuitry with its physical environment, such as transients in power supply, crosstalk, electromagnetic interferences, temperature variation, α and cosmic radiation, etc.

Recent studies [1] [5] point out that advances in semiconductors technologies are greatly increasing the rate of occurrence of transients faults in deep-submicron manufactured systems. Furthermore, new and more complex transient fault models have to be considered as the size of the transistors shrinks.

[1] http://www2.laas.fr/DBench/index.html

Results from [4], summarised in Figure 2.2, settle the set of fault causes and mechanisms that lead to the definition of transient fault models considered representative of new deep-submicron technologies.

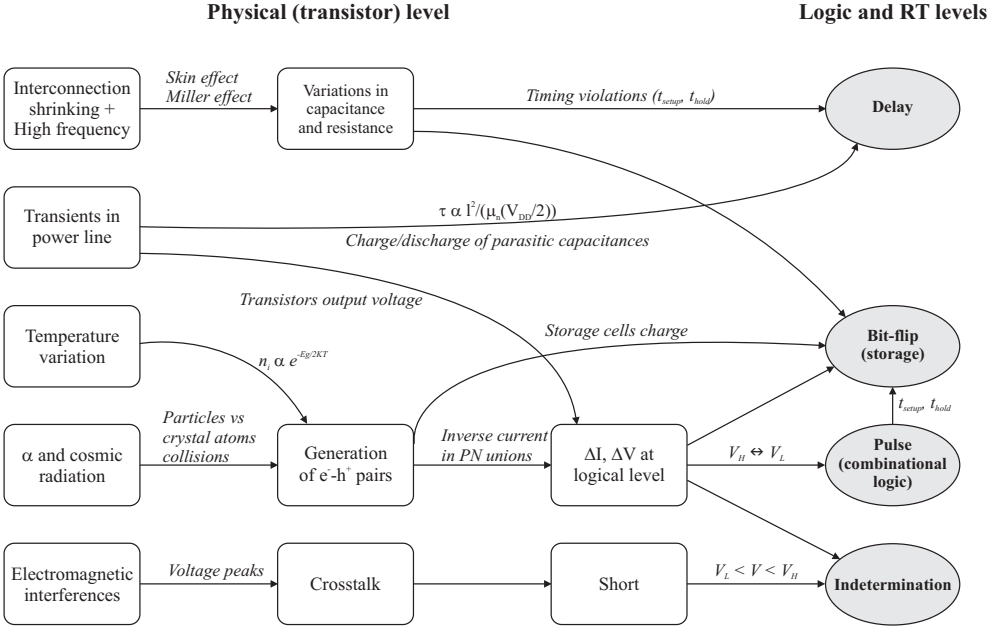

Figure 2.2: *Causes and mechanisms of transient fault models.*

The transient fault models that can be extracted from this study are:

- **Bit-flip**. This is one of the most used fault models. It models the occurrence of a Single Event Upset (SEU) that reverses the current logic state of a memory element, such as registers, latches and memory cells.

  This is a particular transient fault, since it does not present an associated duration. It remains in the system until the affected bit is rewritten by the normal operation of the system.

- **Pulse**. It models the occurrence of a Single Event Transient (SET) in the combinational logic of the circuit. The logic state of the combinational element is reversed for a short period of time, resuming its proper behaviour afterwards.

  Although its effect is similar to that of *bit-flips* into sequential logic, as dealing with combinational logic, the logic state induced by the fault is usually reverted after the fault disappears from the system.

- **Indetermination**. This model assumes that the affected element will present an undetermined voltage value, between the high- and low-level thresholds of a given technology, for the duration of the fault. This results in that logic element holding an undetermined logic state.

- **Delay**. It models a modification, usually an increase, in the propagation delay of the circuit. This may affect the fault-free behaviour of the system by, for instance, causing violations of registers' set-up and hold times or delaying a signal beyond registers' capture window.

Hence, although classical transient fault models, like *bit-flip*, are still valid, they are not longer enough to evaluate the dependability of modern systems.

2.3 Permanent Faults

Permanent faults are due to irreversible physical defects in the circuit. They usually appear as a result of the manufacturing process or the normal operation of the system. In this latter case, sometimes they initially reveal as intermittent faults until some long term wearout mechanisms cause the occurrence of a permanent fault.

Late studies [1] call attention to the fact that new semiconductor technologies are not increasing the rate of occurrence of permanent faults as much as that of transient ones. Improvements in the manufacturing process are limiting the impact of the reduction of the transistors' size on the likelihood of occurrence of permanent faults.

However, the probability of occurrence of intermittent faults is increasing in deep-submicron manufactured systems. Many of these faults will, in the long term, manifest as permanent ones thus increasing the rate of occurrence of permanent faults due to wearout mechanisms.

Results drawn from [4], shown in Figure 2.3, determine the set of fault causes and mechanisms that support the definition of permanent fault models considered representative of new deep-submicron technologies.

The considered permanent fault models are:

- **Stuck-at**. This is the most widely used permanent fault model. It assumes that the output of the targeted logic element is bound to a determined logic value regardless the evolution of its inputs.

  That fault model presents two different forms named *stuck-at-0* and *stuck-at-1*, depending on the logic state the affected element is bound to.

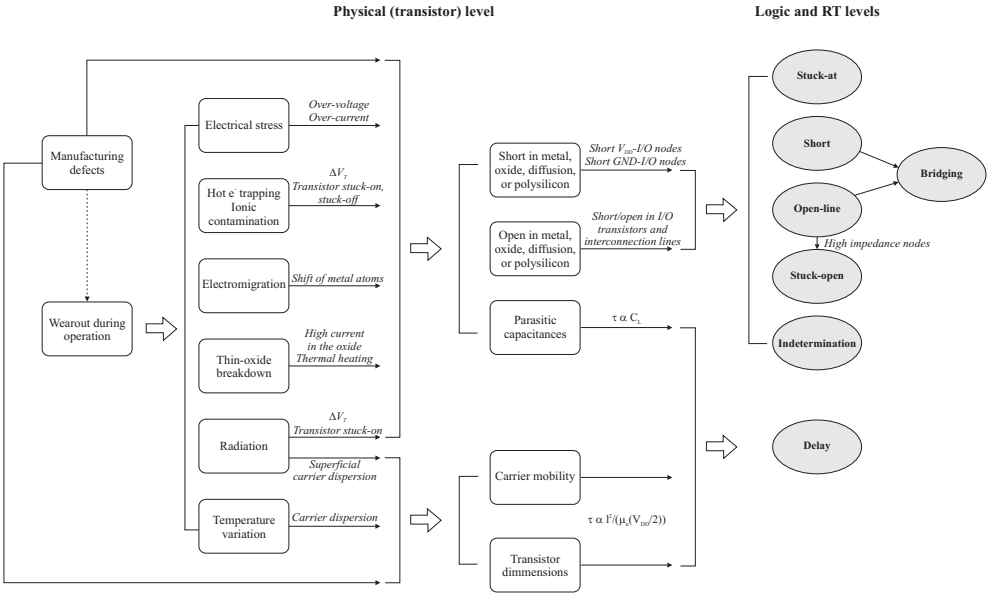

Figure 2.3: *Causes and mechanisms of permanent fault models.*

- **Stuck-open**. This is a special fault model related to floating high-impedance nodes in CMOS technology. The affected element holds its previous logic value and, after the output parasitic capacitances are discharged by leakage currents (*retention time*), its state is bound to a low-logic level (`'0'`).

- **Indetermination**. As in the case of transient *indeterminations*, it models an undetermined voltage value between the high- and low-level thresholds that results in an undetermined logic value. The only difference is related to the duration of the fault.

- **Delay**. This is another fault model that is also found as a transient one. It represents an increase in the propagation delay of the circuit.

- **Short**. It models a modification in the routing of the system that results in the short-circuit (interconnection) of two different lines of the circuit. This may affect the fault-free behaviour of the system depending on the logic states driven into the targeted lines.

- **Open-line**. *Open-lines* result from problems in the routing of the circuit that break up a line, which interconnected different logic elements, into two separated segments. This may affect the behaviour of the system since elements connected to the affected line are no longer being driven.

- **Bridging**. Although *bridging* has been used along the years to refer to a short-circuit between two lines of the circuit, that notion has been labelled here as *short*. In this work, as stated in [6] and [7], *bridging* models the occurrence of a special combination of *short* and *open-line*.

  According to these works, there exists four different *bridging* cases that can result from the combination of a *short* and an *open-line* affecting two lines of the circuit. This combination can be modelled as follows: one of the lines is affected by an *open-line* and then one of the resulting segments is connected to the other line (*short*). The different scenarios derived from the existing combinations are depicted in Figure 2.4.

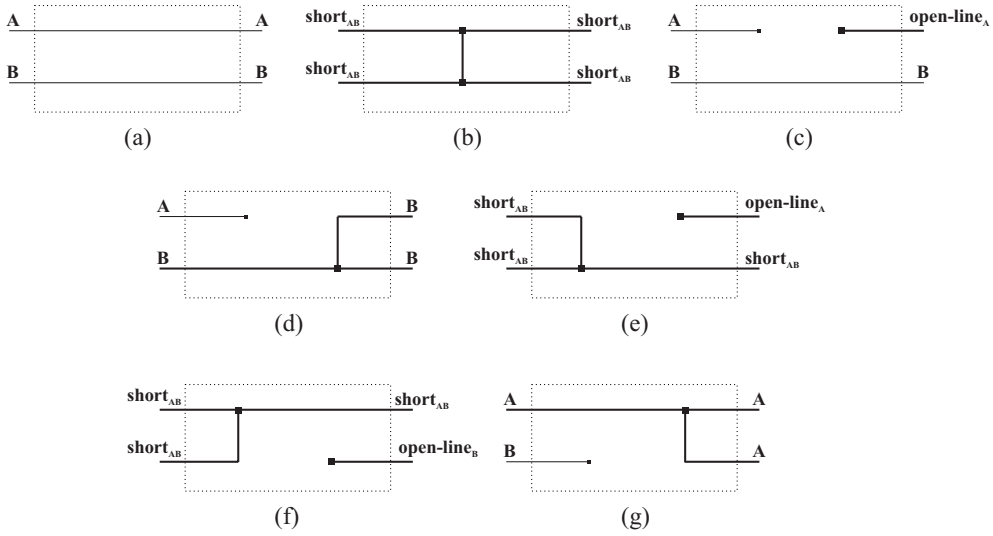

Figure 2.4: Bridging *possibilities as the combination of a* short *and an* open-line. *Two connecting lines of a circuit (a) may be affected by a* short *(b) or an* open-line *(c). The combination of these two fault models may lead to four different situations (d) (e) (f) and (g), in which one of the lines is affected by the* open-line *and one of the resulting segments is connected to the other considered line (*short*).*

Although not significantly increasing the overall rate of occurrence of permanent faults, the number and kind of existing permanent fault models have expanded beyond classical ones. Hence the importance of studying the dependability of modern systems in the presence of new and more complex permanent faults considered representative of modern semiconductor technologies.

2.4 Conclusions

The likelihood of occurrence of faults, specially transient ones, in deep submicron manufactured systems is increasing as the size of the transistors shrink. Although permanent faults due to the manufacturing process are less and less important, permanent faults related to the wearout of components due to normal operation, and intermittent faults finally manifesting as permanent ones, are also becoming a major concern in semiconductor technologies. On the other hand, the advent of nanoelectronic devices will greatly increase the occurrence of defects in the near future. Hence the importance of studying the behaviour of systems in the presence of representative hardware faults.

A representativeness study of how faults occurring at transistor level manifest at logic and RT levels, have determined the set of fault models considered representative of hardware faults in deep submicron technologies. These fault models, which will be used along this thesis, are summarised in Table 2.1.

Table 2.1: *Fault models considered representative of deep-submicron manufactured systems.*

Fault model	**Fault duration**	**Description**
Bit-flip	Transient	Reverses the logic state of a memory cell
Pulse	Transient	Reverses the logic state of a combinational logic element for the duration of the fault
Indetermination	Transient / Permanent	Undetermined logic value between the low- and high-level thresholds
Delay	Transient / Permanent	Increases the propagation delay of a line
Stuck-at	Permanent	Fixes the logic value of a logic element
Stuck-open	Permanent	Fixes the logic value of a logic element for a retention time, and to '0' afterwards
Short	Permanent	Short-circuits two lines
Open-line	Permanent	Splits a line into two parts
Bridging	Permanent	Special combination of open-line and short

Chapter 3

FPGA-based Fault Injection

Although originally conceived with prototyping purposes, FPGAs have been recently proposed as a means to efficiently accelerate dependability assessment via model-based fault injection. This methodology, known as fault emulation*, achieves the goals of enabling the early, fast, and low cost evaluation of new deep-submicron manufactured systems in the presence of faults.*

This Chapter discusses the existing fault emulation approaches, based on Compile- and Run-Time Reconfiguration techniques, with the purpose of identifying their main benefits and drawbacks to determine which is the most suitable approach for the injection of faults into large and complex systems. The need of a generic FPGA architecture to sustain the development of new methodologies for emulating the considered hardware fault models using the selected approach is also addressed in this Chapter.

3.1 Introduction

When *Field-Programmable Gate Arrays* (FPGAs) reached the market they were considered as one more of the many configurable devices available at that time. They were destined to implement combinational logic and act as *glue logic* for low volume systems due to their very limited speed and capacity, and their high cost per unit. Along the years FPGAs have evolved into more complex devices [8] which not only integrate a large amount of programmable logic but also a number of embedded components such as memory blocks, multipliers and microprocessors cores. Nowadays, they are used as targets for the final implementation of systems as well as for prototyping purposes.

There exists a wide variety of FPGA's technologies, such as *antifuse-* (no reprogrammability, but high speed), *flash-* (non-volatile), and Static Random Ac-

cess Memory (SRAM)-based devices (fast reprogrammability, larger devices). Currently, the market share is dominated by SRAM-based FPGAs due to their in-circuit fast reprogrammability and their capacity to hold larger designs. SRAM-based FPGAs consist of a grid of configurable logic elements that are controlled by means of different SRAM cells. The configuration of these FPGAs can be easily modified by programming and reprogramming their SRAM configuration memory. This can be achieved in-circuit, i.e., without extracting the device from the circuit it is integrated into. Therefore, SRAM-based FPGAs offer a great flexibility and can be successfully used in reconfigurable computing applications.

One of the main application domains of FPGA-based computing is the *Application Specific Integrated Circuit* (ASIC) logic emulation. The logic validation of an ASIC is a critical stage in its manufacturing process. It tries to determine whether the current implementation of the logic circuit fulfills the requirements specified at the design stage. Three different approaches [9] have been developed to perform the logic validation of an ASIC:

- **Hardware prototyping.** It consists in obtaining a physical implementation (prototype) of the circuit to validate from which representative results can be extracted. Experiments are rapidly executed since the speed of the prototype is usually very close to that of the final system. However, that prototype is only available in the last stages of the development cycle, when correcting any detected error causes vast costs.

- **Software simulation.** Instead of a prototype, this technique makes use of a model of the functional and/or electrical behaviour of the implemented circuit which is simulated on a computer. In this way, the system can be validated along all the stages of its development cycle, reducing the cost of fixing any error. However, the time required to simulate very complex models is so high that this technique lacks of applicability in some cases.

- **Logic emulation.** It consists in the use of FPGAs to implement the model of the system [10]. Logic emulators [11] are somewhere between hardware prototypes and software simulators. The model of the system is emulated much faster than in the case of software simulation, although it cannot rival the speed of ASIC prototypes. Moreover, it is easier, cheaper and faster to implement the model of the system on an FPGA than building its prototype. They are also more flexible and easy to use. All these reasons make FPGAs well-suited devices for the logic validation of circuits.

Nevertheless, the high rate of faults occurrence in new semiconductor technologies makes these techniques insufficient for the validation of systems. It is not only necessary to study the functional behaviour of the system, but also its behaviour in the presence of representative faults as ultimate cause of system's failure.

The impossibility of observing systems on the field to get statistical data makes *fault injection* [12] a very valuable methodology in the validation process. Fault injection, which consists in the deliberate introduction of faults into a system, can be used to assess its dependability, and possibly complement other approaches like modelling that lack both applicability and accuracy when dealing with complex systems. Although there exists a wide range of different approaches, they are typically classified into two big families named *prototype-* and *model-based fault injection* techniques.

- **Prototype-based fault injection techniques.** These methodologies, which require a prototype to introduce the faults into the system, are further divided into *Hardware Implemented Fault Injection* (HWIFI) and *Software Implemented Fault Injection* (SWIFI) [13] [14].

  HWIFI techniques use additional hardware to introduce physical faults like short circuits or electro-magnetic interferences into the target system. Several techniques and tools have been developed with this purpose: MESSALINE [12], RIFLE [15] and AFIT[1] [16] are representative tools of pin-level fault injection techniques; [17] describes the use of electromagnetic interferences for fault injection; heavy-ion radiation was used in [18] and by FIST [19]; FIMBUL [20] suggested the use of built-in scan-chains; finally, [21] proposed the use of a laser beam for fault injection. The required extra hardware usually increases the cost of applying HWIFI techniques and some of them may even damage or interfere with the system under test.

  SWIFI techniques are usually a low-cost alternative to HWIFI since they do not require so expensive hardware and, furthermore, they can target operating systems and applications. FERRARI [22], Xception [23] and MAFALDA [24] are well-known tools that make use of SWIFI techniques. INERTE[1] [25], which makes use of the Nexus [26] standard debugging interface to inject faults into the system, was built in the frame of the Dependability Benchmarking project[2] [27] to tackle with the dependability benchmarking of engine control applications in automotive embedded

[1] This tool was developed by the Fault Tolerant Systems research Group (GSTF) of the Universidad Politécnica de Valencia (UPV) in Spain

[2] http://www.laas.fr/DBench/

systems. Although SWIFI is a flexible approach, it cannot target locations that are inaccessible to software, like hidden registers, and may disturb the workload running on the system under test.

Even though the use of prototypes causes the experiments to be executed very rapidly, these techniques can only be applied in the last stages of the development cycle, thus increasing the cost of fixing any error in the design.

- **Model-based fault injection techniques.** These techniques [28] partially solved that particular problem by injecting faults into a model of the system. Since only a model is required and not a final prototype, these techniques allow for the early validation of the system, thus reducing the cost of fixing any error in the design. Tools like MEFISTO [29], VERIFY [30] and VFIT[1] [31] can be used to inject faults into VHDL (Very High Speed Integrated Circuits Hardware Description Language) models. These techniques clearly follow the same approach as software simulation for logic validation and thus the simulation of complex models required an enormous amount of time.

It is to note the resemblance in the methodologies used in fault injection and logic validation techniques. Hence, why not use FPGAs to implement the model of the system and thus accelerate the execution of model-based fault injection experiments? It should also be easier and faster than prototype-based fault injection techniques (HWIFI and SWIFI). That technique, which presents the same basic characteristics than logic emulation for logic validation, is named *fault emulation*.

Understanding fault emulation requires some basic knowledge about how FPGAs work. Section 3.2 defines the basic architecture of FPGAs and presents the common design flow for configurable logic applications using programmable devices. The two different existing approaches for fault emulation, *Compile-* and *Run-Time Reconfiguration*, are discussed in Section 3.3 to determine the best suitable approach for dealing with the emulation of hardware faults representative of new deep submicron manufacture systems. A generic FPGA architecture is defined is Section 3.4 to enable the future definition, along this thesis, of new generic methodologies for the emulation of the considered hardware faults. Finally, Section 3.5 concludes the Chapter.

3.2 Basic FPGA architecture and design flow

SRAM-based FPGAs basically consist of a two-dimensional array of logic elements which can be programmed to implement the circuit logic. Those logic elements are interconnected by means of some programmable routing. The configuration memory of the FPGA controls the current configuration of each of these elements, being responsible then for implementing the desired circuit. This fairly simple architecture [32], depicted in Figure 3.1, has evolved along the years, mainly by the addition of some more elements like RAM memory blocks, multipliers or microcontroller cores into the fabric logic of the FPGA.

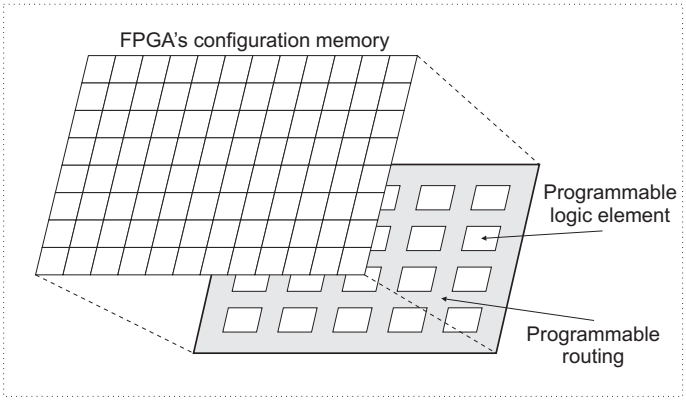

Figure 3.1: *Basic FPGA architecture.*

The common design flow for implementing the model of a system into an FPGA comprises a number of successive processes that must be fulfilled. Although each FPGA manufacturer refers to the same processes with different names, the whole design flow is very similar for all of them.

This flow, which is shown in Figure 3.2, consists in the following steps:

1. **System design.** When entirely designing a new system from scratch or even when building a system from other components, it is necessary to specify a functional model of that system. This model, commonly specified by means of some *Hardware Description Language* (HDL) such as Verilog [33], VHDL [34] or SystemC [35], is the starting point of any FPGA design flow. System models may be described in different abstraction levels, but since the goal of these models is to be implemented on an FPGA, their functional structure is usually described in the *Register-Transfer Level* (RTL).

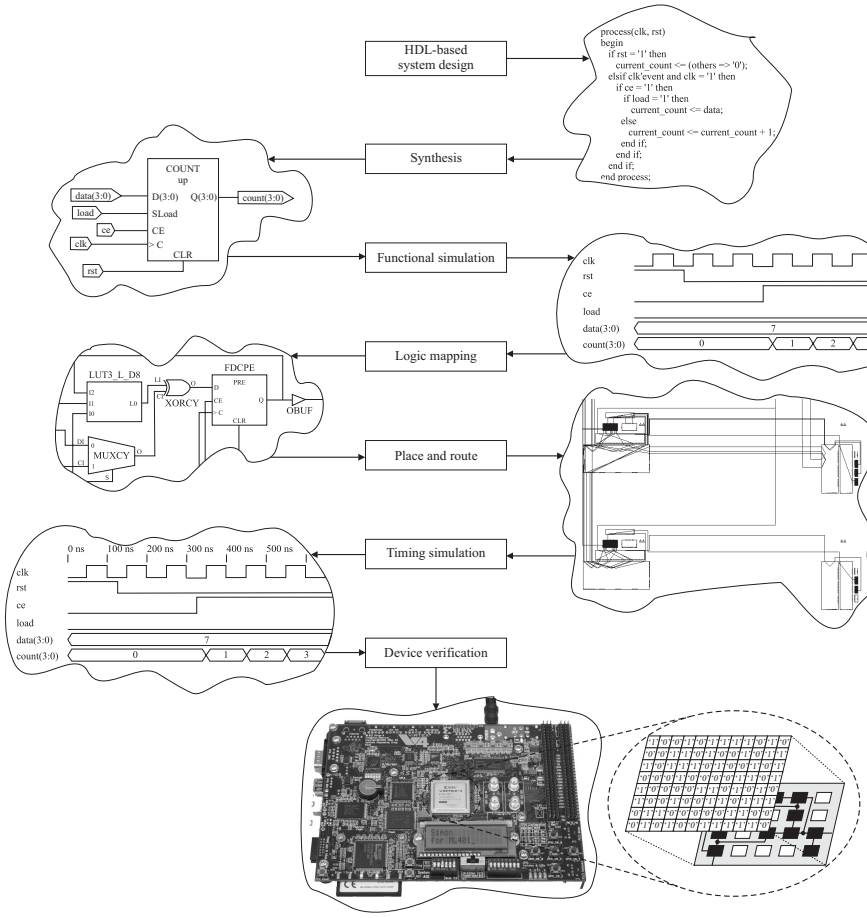

Figure 3.2: *Common FPGA design flow.*

2. **Synthesis.** The logic synthesis is a process in which the model of the system is analysed to extract the logic elements that have been specified in the model and their interconnection. This is a critical step since a bad synthesiser or a poorly described model may cause the inference of a system completely different from the one intended. Although each FPGA manufacturer provides its own synthesiser, other companies also commercialise their own products such as Leonardo Spectrum from Mentor Graphics[3] or Synplify from Synplicity[4].

[3] http://www.mentor.com/
[4] http://www.synplicity.com/

3. **Functional simulation.** Once the model of the system has been synthesised, it is very advisable to simulate the result of that synthesis to determine whether it presents the desired functionality. Several different simulators are available in the market, being ModelSim[5] from Mentor Graphics one of the most recognised. If any error is detected, it is necessary to modify the model of the system and iterate again through the previous steps, in other case, the implementation phase begins.

   Up to this step, unless the model of the system includes some constructs for a specific FPGA family, the results of the synthesis process are device independent and, therefore, the system can be implemented in any reconfigurable device. From this point and on, the results of each step will be dependent on the selected FPGA architecture.

4. **Logic mapping.** The logic mapping process maps the generic logic elements that have been extracted by the synthesis to the actual resources of the selected FPGA that will be necessary to implement that functionality. For instance, implementing a four-variable combinational function will usually involve one function generator, while implementing a four-bit register will require four flip-flops. Hence, this process provides the total number of FPGA logic resources that will be necessary for the implementation of the circuit and their relationship.

5. **Place and route.** This step comprises two different although highly intertwined processes: placement and routing. A detailed description and pseudo-code for these algorithms can be found in [32].

   The placement process consists in selecting which of the free logic resources of the FPGA will be used to implement the desired circuit. All the resources that have been determined by the logic mapping must be distributed throughout the FPGA, usually optimising either the area or the speed of the system. This distribution is achieved by means of a *Simulated Annealing* algorithm [36] [37]. Firstly, all the elements are *randomly* distributed among the free resources of the FPGA. After that, these elements randomly swap locations whether this change optimises a given cost function or, with a certain probability, in case the change does not optimise the cost function. As can be seen, this is not a deterministic process and thus several executions could be needed to minimise as far as possible the cost function.

[5] http://www.model.com/

The routing process tries to perform the required interconnections among all the already placed logic elements on the FPGA. Since FPGAs routing architecture is usually represented as a directed graph, the routing process consists in finding a path between the nodes that represent the pins of the blocks to be connected. Paths must be as short as possible, use fast routing resources for critical connections and not use routing resources required by other connections.

Those processes may fail as a result of great congestion on the FPGA routing, for instance. Usually, these problems are solved by using a larger device (with more programmable resources), or by optimising the model for a better synthesis and, consequently, a better use and allocation of the available resources.

Finally, this step provides a file that can be downloaded onto the FPGA configuration memory to configure the programmable device to functionally emulate the behaviour of the desired system.

6. **Timing simulation.** Before testing the implementation of the system's model on the real device, it is advisable to simulate the result of the place and route process. This is not only a functional simulation, but it also includes the timing and delays for all the logic and routing resources of the FPGA that have been used in the implementation of the circuit. Any problem regarding the delay at some specific routing lines or paths must be corrected by placing and routing the design again under more restrictive conditions. In case that no optimisation could be performed at that level, an architectural change should be considered.

7. **In-device verification.** The last step is the verification of the implementation on the actual FPGA. A correct workload execution on the prototype board will grant the success of the final implementation of the system.

Once the basic architecture and common design flow of FPGAs have been described, it is possible to tackle the origins and current state of the art of fault emulation, identifying the most relevant approaches and how they influence the presented design flow with fault injection purposes.

3.3 Fault emulation

Nowadays, there exists different commercial systems, such as the DN8000K10 from *The Dini Group*[6] and ET5000K10M from *Emulation Technology*[7], that enable the rapid prototyping and logic validation of ASICs. These systems, named *logic emulators*, usually consist of one or several motherboards which hold a certain number of FPGAs depending on the complexity of the system to be implemented. The use of logic emulation principles and tools to speed-up model-based fault injection experiments is known as *fault emulation*[8].

One of the first attempts to use logic emulators with fault-injection purposes was proposed in [38] under the name of *serial fault emulation*. A software tool for implementing system models onto FPGAs was used to generate the configuration file of the fault-free circuit. This file could be used to debug the circuit and obtain the expected values for the considered outputs. That same tool generated a new configuration file for each fault to be injected into the system. These files reconfigured the logic emulator to emulate the behaviour of the system in the presence of the particular injected fault.

The fault injection process consisted in the following steps:

1. Execution of the fault-free circuit until the injection time was reached.
2. Reconfiguration of the logic emulator to inject the fault.
3. Registers initialisation to restore the previous state of the system before the reconfiguration.
4. Emulation of the faulty circuit.
5. Reconfiguration of the logic emulator to remove the fault because of either emulating transient faults or reaching the end of the experiment.

Thus, the *serial fault emulation* methodology involves synthesising and implementing a different model for each fault being injected into the system, and reconfiguring the FPGA to inject and delete each considered fault.

These first approaches were mainly aimed at obtaining the *test coverage* (fault grading), which measures the efficiency of test vectors when detecting

[6] http://www.dinigroup.com/

[7] http://www.emulation.com/

[8] Some authors consider that SWIFI techniques emulate the occurrence of faults as they inject software and hardware errors. Our view of fault emulation is the use of logic emulators to emulate the manifestation of faults into the implemented circuit according to a given fault model.

defects in silicon-manufactured systems. It is usually represented as the percentage of faults that can be identified by that set of test vectors.

Although FPGAs accelerated the execution of the fault emulation process, two were the main aspects that limited the attainable speed-up.

- **Model synthesis and implementation**. The time dedicated to the implementation of medium- to high-complexity models ranges from few minutes to some hours. As *serial fault emulation* requires a new implementation for each considered fault, it can lead to so long global implementation times that this methodology might result impracticable.

  Hence, new techniques must be developed to minimise the number of implementations required for the execution of fault injection campaigns. In the best case, the minimum possible number of implementations is just one, the fault-free circuit.

- **Fault injection and deletion**. The injection of each fault under the *serial fault emulation* methodology involves the reconfiguration of the logic emulator to perform the injection and to delete the fault later. The time devoted to fully reconfigure a typical FPGA, when downloading the reconfiguration file from a PC platform, is usually in the order of tens of milliseconds. Taking into account that logic emulators commonly hold several FPGAs, the global reconfiguration time may greatly exceed the execution time of the experiments on the logic emulator.

  Thus, the interest in developing new techniques to minimise both the number of reconfigurations and the size of the reconfiguration files to be downloaded onto the FPGAs.

At present, two different methodologies intended for the ease of these problems have evolved from the existing approaches for the design of reconfigurable applications [39]. These methodologies, known as *Compile-* and *Run-Time Reconfiguration*, their advantages and drawbacks, are described next.

3.3.1 Compile-time reconfiguration

Compile-Time Reconfiguration is the simplest and most widely used technique for implementing applications with reconfigurable logic. All the different required functionalities of the systems are described in the same model and a number of control signals are used to select the desired functionality. Once implemented on an FPGA, the proper activation of the control signals will cause the circuit to assume the requested functionality.

The first attempts to use that methodology with fault injection purposes were proposed in [40] [41] under the name of *dynamic fault injection*. They consisted in modifying the model of the system to include a global circular shift register to control the fault activation signals. The look-up tables that implement the combinational logic of the circuit are duplicated and, as shown in Figure 3.3, to implement the fault-free and the faulty behaviour of the circuit. A fault activation signal, which is shared by a set of look-up tables, controls the multiplexer that selects between the two replicas. In that way, faults are injected by changing the contents of the affected look-up table in each set and recompilation is completely avoided.

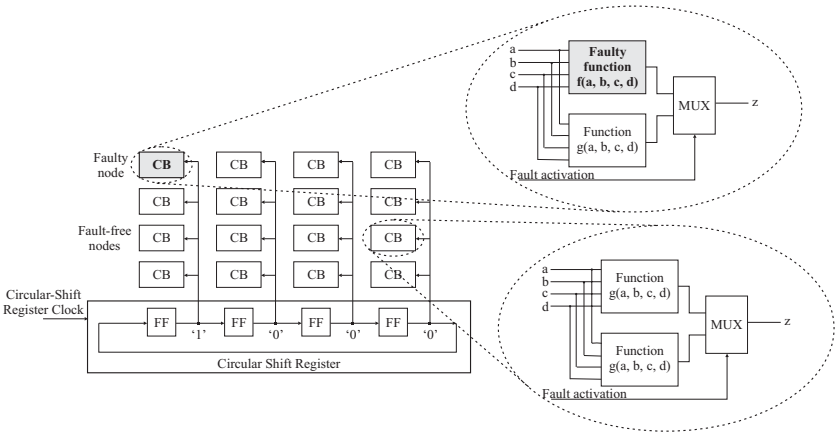

Figure 3.3: *Dynamic fault injection methodology.*

This technique was improved in [42] [43] by converting the original model of the system into a *fault injectable circuit*. The required logic to inject all the desired faults and to control the process is included in the model and implemented into the FPGA. Hence, no reconfiguration is needed to inject any fault, just activating the proper control signal.

Three kinds of *fault injection elements* were proposed for the injection of faults into the circuit. These elements are inserted into the selected points of the circuit, as shown in Figure 3.4, and their control signals are connected by means of a *fault injection scan chain*. Faults can be easily injected into the circuit by shifting the data in the scan chain.

In order to reduce the number of flip-flops required for the scan chain implementation, and thus the size of the circuit, [43] suggests the implementation of a decoder-based fault injectable circuit. This approach is further extended in [44] by using row and column decoders to select the fault to inject.

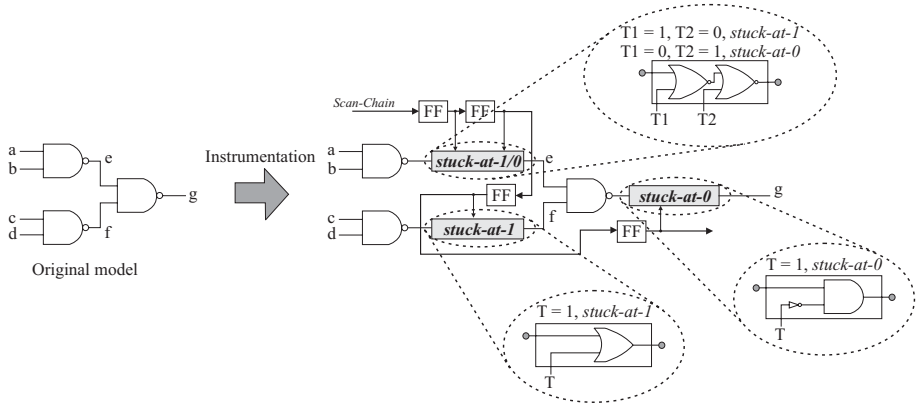

Figure 3.4: *Implementing scan chain-based fault injectable circuits.*

These techniques were combined in a fault injection tool named *defin* (***de****multiplexer based* ***f****ault* ***in****jection*) [45] that was developed at IST/INESC-ID[9] in Portugal. A *hybrid shift register-demultiplexer approach* was considered to find a trade-off between the resources required to implement the demultiplexers and their easy use.

That methodology was also applied at TIMA[10] (Techniques of Informatics and Microelectronics for computer Architecture Laboratory) in France to inject faults into a *fuzzy* microcontroller [46]. Each flip-flop of the circuit was substituted by a custom flip-flop which could modify its value by activating an external signal.

Probably, the most renown fault injection tool that follows the Compile-Time Reconfiguration approach is *FIFA* (***F****ault* ***I****njection by means of* ***FPGA***), developed by the Electronic CAD and Reliability Group[11] of the Politecnico di Torino in Italy. The gate-level model of the system is instrumented following a scan chain-based approach [47] [48]. The control circuit, that can be seen in Figure 3.5, consists in two different modules:

- **Mask Chain**. Each flip-flop of the circuit is associated to a bit of the Mask Chain register, which can be initialised by means of the *scan_in* signal. The activation of the *inject* signal will cause the injection of a fault in those flip-flops whose associated bit holds a logic `'1'`. It is also possible to load the value of the flip-flop and read the state of the system via the *scan_out* signal.

[9] http://www.inesc-id.pt/
[10] http://tima.imag.fr/
[11] http://www.cad.polito.it/

- **Masking logic**. Combinational logic in charge of possibly inverting the output of the combinational circuitry associated to the flip-flop.

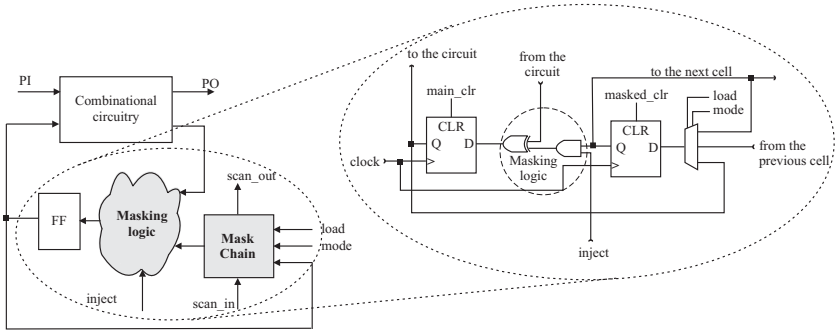

Figure 3.5: *Model instrumentation by using the* FIFA *tool.*

FIFA was optimised to enable the injection of faults into VLSI systems [49] [50]. The analysis of the system's behaviour in the presence of faults was also optimised to reduce the execution time of the experiments. Several circuits from the set of benchmarks ITC'99 [51] were used to validate this version of the tool.

This methodology was later adapted [52] to assess the dependability of microprocessor-based systems implemented as a *System on a Chip* (SoC). The microprocessor's core is instrumented, as shown in Figure 3.6, by means of a number of new defined modules, such as *Memory Stub logic*, *Bus Stub logic*, *Masking logic* and *Fault Injection Bus*.

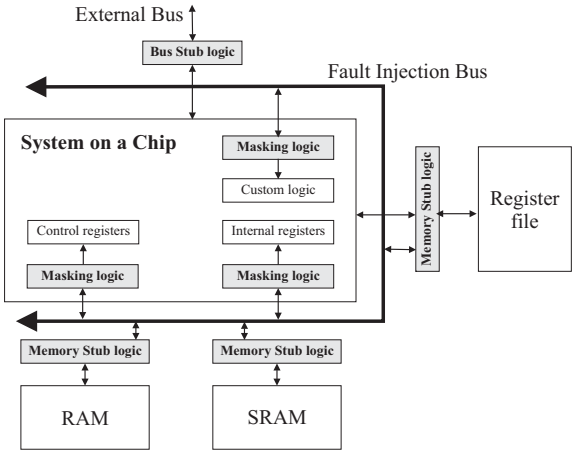

Figure 3.6: *Instrumentation of a microprocessor's core implemented as a SoC.*

The Grupo de Microelectrónica[12] of the Universidad Carlos III in Spain proposed the use of an autonomous FPGA-based emulation system to further improve the attainable speed-up [53]. The main idea is avoiding any communication between the host and the prototyping platform, being the FPGA responsible for applying the stimuli, injecting the faults and checking the output values. The performance and area overhead analysis of several different techniques that follow this approach was presented in [54].

Lately, the Dependable Systems Laboratory[13] of the Sharif University of Technology in Iran has been working on the implementation of a new hybrid fault injector. This tool, named FITSEC (***F**ault **I**njection **T**ool based on **S**imulation and **E**mulation **C**o-operation*), is aimed at obtaining the advantages of software simulation and compile-time reconfiguration while minimising their drawbacks [55]. The application of this methodology to inject faults at transistor-level models was presented in [55] [56].

All those recent Compile-Time Reconfiguration techniques achieve the goal of improving the original *serial fault emulation* approach.

3.3.2 Run-time reconfiguration

Run-Time Reconfiguration applications dynamically reallocate hardware during their execution on an FPGA. Each application comprises *multiple* configurations per FPGA, with each configuration representing a fraction of the application. The execution of the application on the FPGA consists in dynamically reconfiguring the programmable device to implement the required functionality and release those resources that are not needed anymore.

The first attempts to apply this technique with fault injection purposes [57] [58] were proposed by the Qualification of Circuits research group[14] of TIMA in France. The dynamic modification of function generators' contents (see Figure 3.7) was proposed to emulate the occurrence of faults that force the output of function generators and synchronously invert the contents of flip-flops.

A similar approach was presented in [59] to inject *Single Event Upsets* (SEUs) or *asynchronous bit-flips* into the sequential logic of the circuit. The dynamic modification of a flip-flop's state involves determining its current state and configuring the related logic to trigger its set or reset input, accordingly.

[12] http://www.uc3m.es/uc3m/dpto/IN/dpin08/dpin08d.html
[13] http://ce.sharif.edu/∼dsl/
[14] http://tima.imag.fr/qlf/

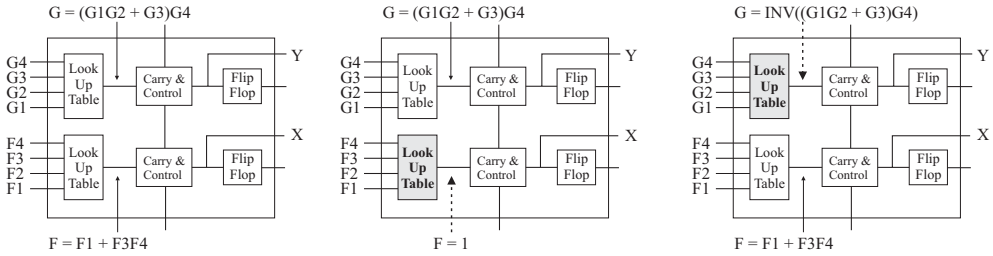

Figure 3.7: *Example of dynamic allocation of resources from the original configuration (a), to set the output of a look-up table (b) and synchronously invert the contents of a flip-flop (c).*

A particular implementation of this methodology was presented at [60] [61], where it appears as a novel and efficient methodology for the early fault injection and dependability analysis of systems. The theoretical study of the time required to reconfigure the FPGA to emulate each one of the proposed fault models is detailed in [62].

In this same line, the Departamento de Ingeniería Electrónica[15] of the Universidad de Sevilla in Spain has been working during the last years in the development of a tool named UNSHADES[16] (*UN*iversity of *S*evilla *HA*rdware *DE*bugging *S*ystem). Although UNSHADES was firstly aimed at debugging VLSI systems by observing its internal state, the last version of that tool, FT-UNSHADES (Fault Tolerant-UNSHADES), makes use of the Run-Time Reconfiguration approach to emulate the occurrence of SEUs in the registers of a VLSI circuit to study its robustness [63].

The Run-Time Reconfiguration approach has also been successfully used by the IST/INESC-ID[17] in Portugal to evaluate the *Built-In Self-Test* (BIST) effectiveness. A detailed explanation of the procedure to assess the BIST quality by means of FPGAs can be found at [64] [65] [66], where a core of the ARM7 microprocessor is evaluated. This approach was implemented by a tool named f^2s (*F*PGA based *f*ault *s*imulator), which was validated by injecting faults into the set of benchmarks ISCAS'89 [67].

[15] http://www.dinel.us.es/
[16] http://woody.us.es/∼aguirre/Web_unshades/index.htm
[17] http://www.inesc-id.pt/

Another application of Run-Time Reconfiguration is the evaluation and test of FPGA-based systems. In that case, the FPGA is not only a means to inject a fault into the model of the system, but it is also the platform where the final system will be implemented on.

The Institut für Technische Informatik[18] of Technische Universität Graz in Austria has been working on the study of fault injection methodologies for FPGA-based systems. Run-Time Reconfiguration is used in [68] to validate self-stabilising systems, which converge to their expected behaviour although they may have transited to a wrong state.

A methodology based on the dynamic reconfiguration capabilities of FPGA is proposed in [69] to test FPGA interconnection resources.

The work presented in [70] describes different techniques to tolerate the occurrence of SEUs in SRAM-based FPGAs. The proposed architectures were evaluated by means of a fault injection tool developed at Los Alamos National Laboratory[19] for Xilinx. That tool individually inverts every single bit of the FPGA configuration memory to emulate the occurrence of such faults.

All those approaches, although with different objectives in mind, successfully use the Run-Time Reconfiguration methodology to accelerate the original *serial fault emulation* approach.

3.3.3 Discussion

Two different techniques, Compile- and Run-Time Reconfiguration, whose control flow can be seen in Figure 3.8, have evolved from the original *serial fault emulation* to improve the execution time of fault injection experiments.

On one hand, the number of synthesis and implementation processes necessary to inject the whole set of desired faults is reduced. The original *serial fault emulation* required to synthesise and implement a new model for each fault to be injected. Currently, the model is instrumented by including several fault injectors that enables the activation of faults in different locations of the model. In that way, a single synthesis and implementation can accomplish the mission of injecting several faults, thus reducing the time required to generate the configuration file for all the desired faults.

On the other hand, the fault injection and deletion processes of recent Compile-Time Reconfiguration techniques are greatly optimised by the instrumentation of the system's model, which now includes the logic to emulate the

[18] http://www.iti.tu-graz.ac.at/
[19] http://www.lanl.gov/

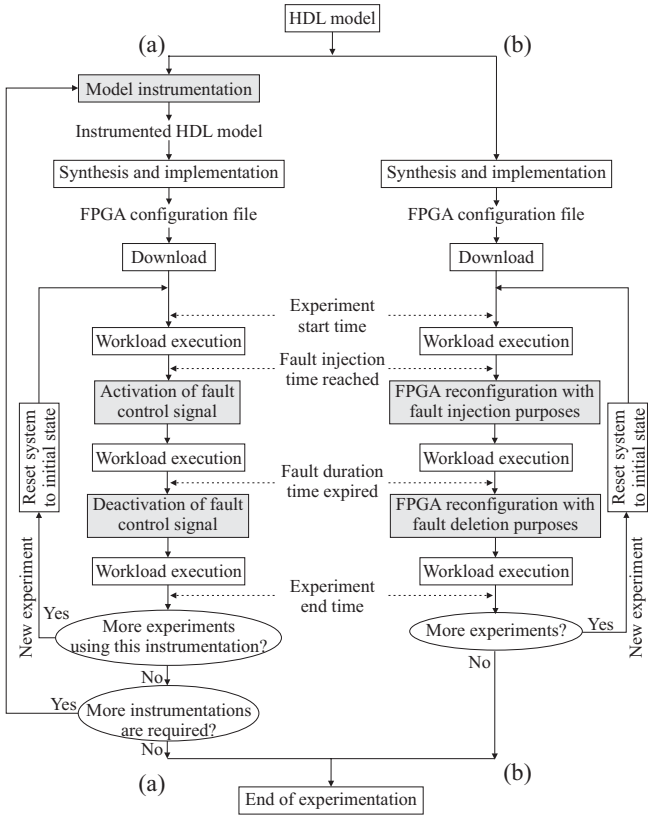

Figure 3.8: *Generic control flow of Compile- (a) and Run-Time Reconfiguration (b) techniques applied to fault emulation.*

behaviour of the system in the presence of faults. Therefore, as no reconfiguration is required to inject or delete the fault, the temporal overhead caused by these processes is negligible.

Although these techniques effectively accelerate model-based fault injection experiments, the size attained by the instrumented model still represents a problem. Complex models are usually difficult to fit into a particular FPGA and, since their size increases after being instrumented, models can easily become unroutable. The common solution to that problem is limiting the amount of additional logic that can be added to a circuit to make it injectable. In this way, it could be mandatory to perform several partial instrumentations of large and complex models to cover the whole set of considered faults. As the synthesis and implementation of large and complex models may last some hours, this will obviously increase the overall experimentation time.

Hence, Compile-Time Reconfiguration can be efficiently used to speed-up model-based fault injection experiments as long as the instrumented model fits the selected programmable device. Otherwise, it would be necessary to divide the model's instrumentation into several partial instrumentations, thus increasing the overall experimentation time. Consequently, this approach is not well-suited for the injection of faults into large and complex models.

The number of required synthesis and implementation processes regarding Run-Time Reconfiguration approaches is reduced to a minimum. As faults are injected and deleted by dynamically allocating the FPGA resources, only the original fault-free implementation of the system is needed. Since synthesis and implementation are the most time consuming processes, a great speed-up can then be achieved due to the implementation of just one model.

Nevertheless, the injection and deletion of each fault involves the reconfiguration of the FPGA to reallocate the resources affected by the fault.

The time consumed in the reconfiguration process is dependent on the reconfiguration capabilities of the selected FPGA [39]:

- **Global reconfiguration**. Every modification on the system's functionality implies the full reconfiguration of the device, regardless the size of the required modification (see Figure 3.9a). This fact increases the time devoted to reconfigure the circuit.

- **Local (also known as partial) reconfiguration**. As the programmable device can be *partially* reconfigured (see Figure 3.9b), only that part of the FPGA affected by the modification must be reconfigured. This approach greatly reduces the fault injection and deletion times since only a small fraction of the device is reconfigured.

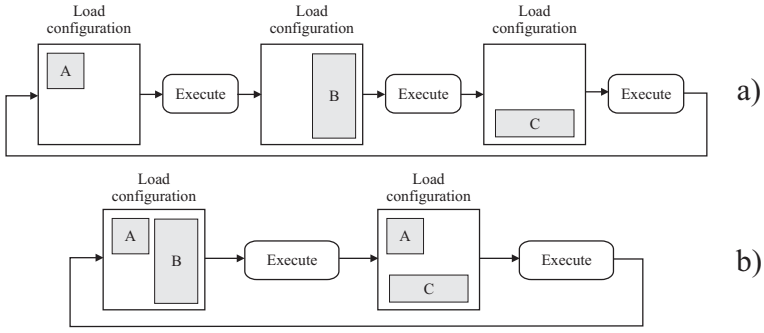

Figure 3.9: *Example of the global reconfiguration (a) and local reconfiguration (b) approaches.*

Currently, the time required to fully reconfigure an FPGA is in the range of tens of milliseconds. Then, even considering the worst reconfiguration scenario, the global reconfiguration approach, the time spent reconfiguring the programmable device is largely counterbalanced by the time saved by implementing the system's model just once.

Hence, Run-Time Reconfiguration can be successfully used to speed-up model-based fault injection experiments specially for large and complex models, when the synthesis and implementation processes could easily take several hours. Consequently, this technique is very well-suited for the injection of faults into complex models.

The main goal of this thesis is to accelerate model-based fault injection experiments for a wide set of hardware fault models considered representative of new deep-submicron manufactured systems. The models of these systems are usually very large and complex, and the more detailed the description of the system, the larger and more complex the model. Thus, Run-Time Reconfiguration seems the best suited methodology in order to obtain the best speed-up ratio when injecting faults into these kind of systems.

This fault injection technique is closely related to the architecture of the selected programmable device. Therefore, the specification of new methodologies for emulating the set of considered fault models by means of FPGAs requires the definition of a generic FPGA architecture that will be used throughout the present thesis. Those new approaches, which will be based on the proposed generic architecture, could be easily adapted later to the particular architecture of any FPGA.

3.4 Generic FPGA architecture

The use of FPGAs for fault emulation requires a more detailed definition of logic elements structure than that presented at [32] and [71], whose concern was limited to the study of FPGAs architectural issues and the definition of FPGA-related fault models, respectively. This study should be based on the different programmable devices currently available in the market.

There exist a large number of FPGA manufacturers such as Xilinx Corp.[20], Altera Corp.[21], Lattice Semiconductor Corp.[22] and Atmel Corp.[23], among

[20] http://www.xilinx.com/
[21] http://www.altera.com/
[22] http://www.latticesemi.com/
[23] http://www.atmel.com/

others. All these manufacturers present a wide variety of high-performance programmable devices, such as the Virtex [72] family from Xilinx (Virtex-II [73], Virtex-II Pro [74], Virtex-4 [75] and the new Virtex-5 [76]), the Stratix [77] family from Altera (Stratix GX [78], Stratix II [79] and Stratix II GX [80]), the LatticeSC (System Chip) [81] from Lattice, and the AT40K family [82] from Atmel.

The generic FPGA architecture that can be extracted from the similarities found among the studied families of FPGAs is based on a two-dimensional matrix of configurable blocks (CBs).

All those configurable blocks are interconnected by means of a network of vertical and horizontal routing segments. The connection among all these segments is established by means of a grid of programmable matrices (PMs).

Also a number of RAM memory blocks are nowadays usually embedded into the fabric logic of the FPGA to allow for the implementation of different applications and systems.

This proposed generic FPGA architecture is presented in Figure 3.10.

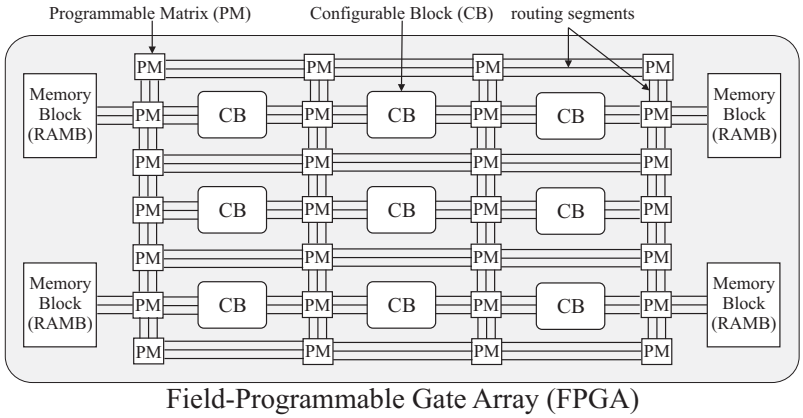

Figure 3.10: *Generic grid-based FPGA architecture.*

The CBs are probably the most important elements when determining the functionality provided by an FPGA, since they are responsible for implementing the logic of the circuit. For that reason, an accurate modelling of a generic CB requires an in-depth study of the architecture of current FPGA's families.

As Figures 3.11 and 3.12 show, representative members of the previously presented FPGA's families essentially share a common structure for their logic elements.

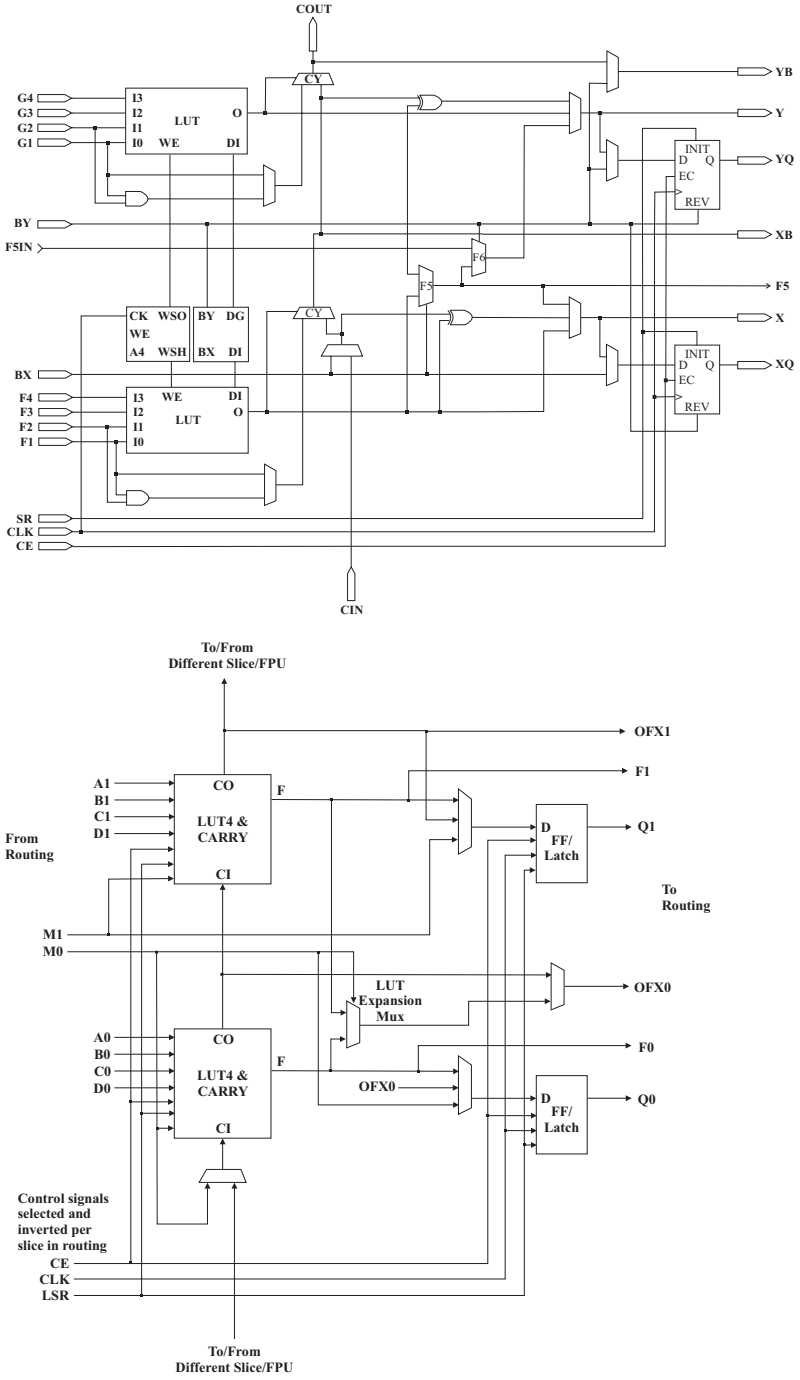

Figure 3.11: *Basic configurable block architecture of most important commercial FPGA families: Xilinx's Virtex (a) and Lattice's LatticeSC (b).*

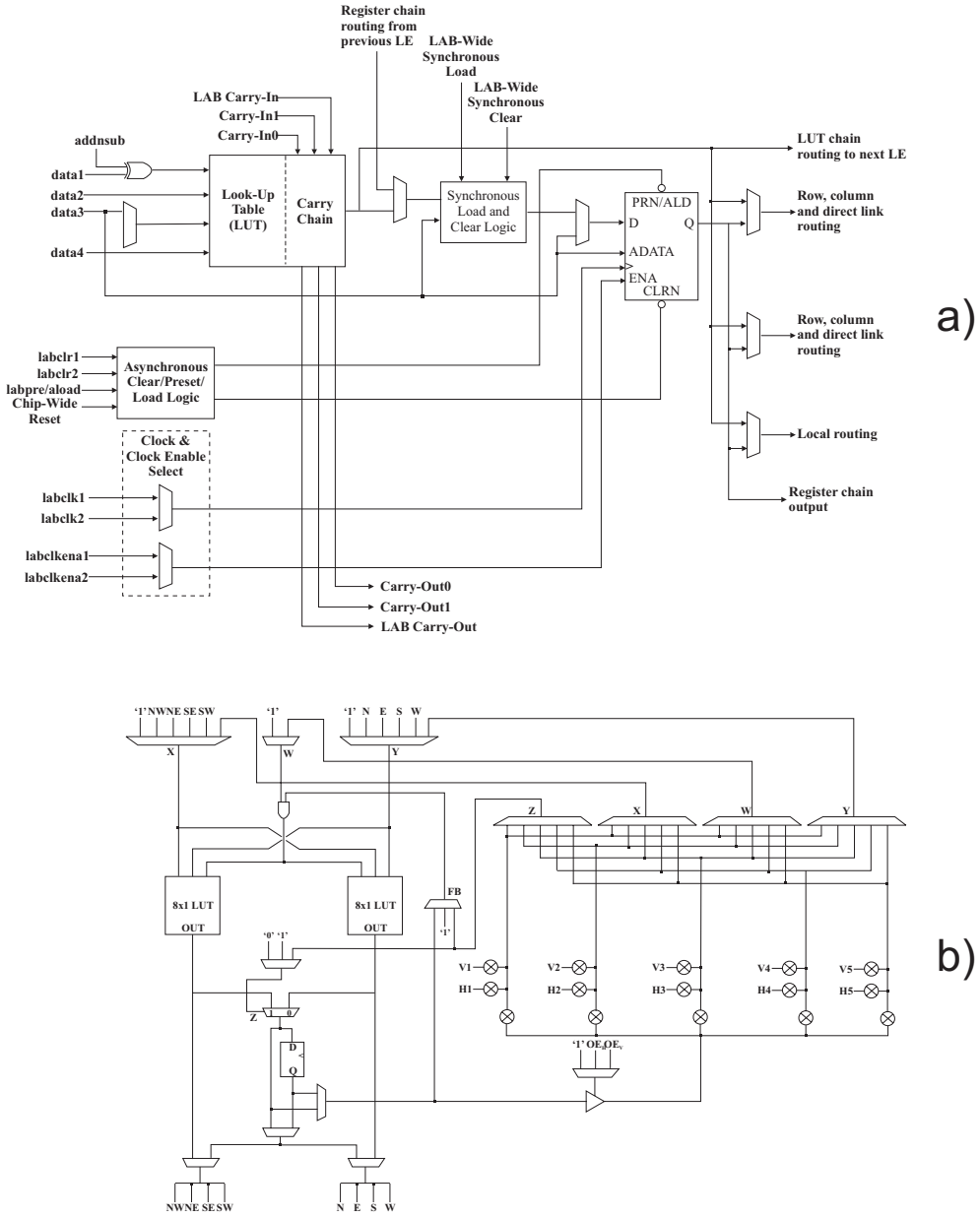

Figure 3.12: *Basic configurable block architecture of most important commercial FPGA families: Altera's Stratix (a) and Atmel's AT40K (b).*

From this study it is possible to conclude that a generic CB, which is in charge of implementing the logic of the circuit, can be modelled as i) a four-input Look-Up Table (LUT) providing the same functionality as a four-variable function for combinational logic; ii) a D-type Flip-Flop (FF) that acts as a storage element for sequential logic; and iii) a number of multiplexers that can modify the functionality provided by the CB.

Each CB presents four different inputs for combinational logic and one for sequential logic. Also, there exists the possibility of obtaining either a sequential or combinational output from the block. The CLK input delivers the clock signal to all the sequential elements of the FPGA. The *Global Set Reset* (GSR) input can be used to trigger the set or reset of all the FFs in all the CBs of the device. Likewise, the *Local Set Reset* (LSR) input can set or reset the state of just the FF it feeds. The structure of the CB, its compounding elements and their possible interconnection is shown in the Figure 3.13.

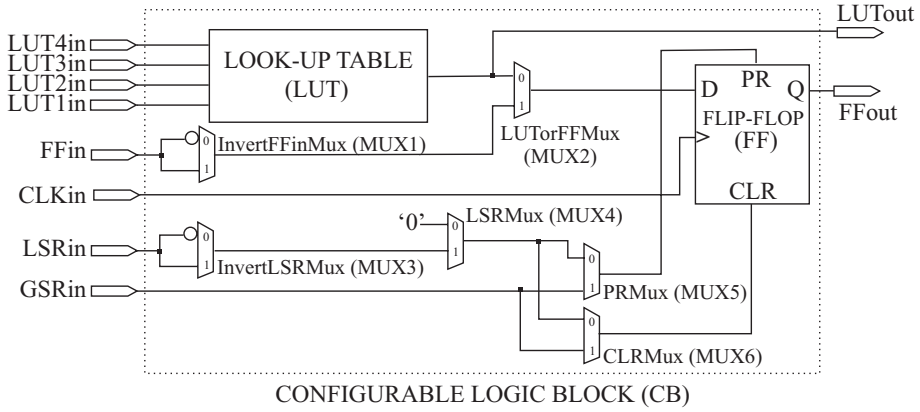

Figure 3.13: *Structure of a generic configurable block.*

The whole set of CBs is interconnected by means of a grid of programmable matrices (PMs). Although the studied FPGA's architectures present different possibilities in terms of interconnection resources, like pass transistors or tri-state buffers, and connectivity capabilities, generic PMs must be kept as simple as possible since dissimilar architectures will simply lead to different routing problems.

Then, a generic PM consists of a number of pass transistors that can be turned on or off to establish a connection between the vertical and the horizontal communication channels of the FPGA. Any line segment may connect to segments in the four possible directions (north, south, east, and west) when reaching an intersection at a PM. Figure 3.14 depicts this particular structure.

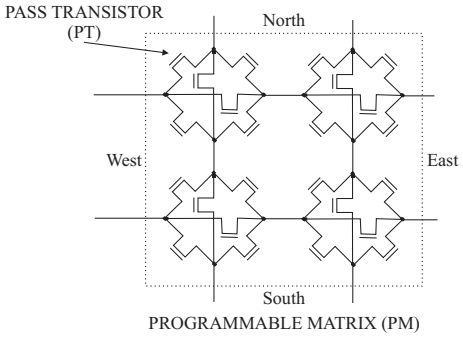

Figure 3.14: *Structure of a generic programmable matrix.*

The configuration memory of the FPGA which, for simplicity, is not depicted in Figure 3.10, controls the behaviour of all the reconfigurable elements.

- **Look-Up Tables**. These elements, which usually describe a 4-input combinational function, are implemented as a 16x1-bit memory. Since their contents are directly represented by some bits in the configuration memory of the FPGA, they can be read and written straightforwardly.

- **Flip-Flops**. The D-type FFs are 1-bit storage elements. The stored information is mapped to 1 bit of the FPGA's configuration memory and, although it can be read, it cannot be directly modified. The only way of changing the current state of a FF is by normal operation or by triggering the set or reset signals of the precise FF.

- **Multiplexers**. The control bit of these elements enables the first input (`'0'`) or the second one (`'1'`) to drive the output of the multiplexer in question. This bit, which corresponds to another bit in the configuration memory of the FPGA, can be directly read and written to change the configuration of the multiplexer.

- **Pass transistors**. These elements are in charge of interconnecting the routing segments of the FPGA. The control bit of a pass transistor can be set to `'1'` to turn it on, thus allowing the communication between two routing segments, or to `'0'` to turn it off, disabling then the connection. This bit, which is mapped to a bit in the FPGA's configuration memory, can be read and written directly for each pass transistor.

- **Memory blocks**. These synchronous blocks store information arranged in rows and columns. All this information is mapped to some bits of the configuration memory of the FPGA and, in opposition to what happened in the case of FFs, they can be read and written straightforwardly.

Therefore, the dynamic resources reallocation, on which Run-Time Reconfiguration approaches rely, is performed by reading and writing the suitable bits of the FPGA's configuration memory.

A number of operations can be defined, according to the proposed architecture, to access (read and write) this configuration memory and, therefore, obtain or modify the current configuration of the element associated to those memory bits. The basic operations that are available to achieve this goal are listed in Table 3.1.

Table 3.1: *Basic operations on the configuration memory of the FPGA to dynamically reallocate its configurable resources.*

Element	**Read operation**	**Write operation**
Look-Up Table L (LUT_l)	$LUT_l()$	LUT_l(new contents)
Flip-Flop F (FF_f)	$FF_f()$	-
Multiplexer M ($MUX[1\text{-}6]_i$)	$MUX[1\text{-}6]_i()$	$MUX[1\text{-}6]_i$('0') $MUX[1\text{-}6]_i$('1')
Pass transistor P (PT_t)	$PT_t()$	PT_t(off), PT_t(on)
Memory block M ($RAMB_b$)	$RAMB_b$(word, bit)	$RAMB_b$(word, bit, '0') $RAMB_b$(word, bit, '1')

Three auxiliary operations (*trace*, *neighbours* and *route*), which ease the task of dealing with the FPGA's routing elements, are defined in Table 3.2.

Table 3.2: *Auxiliary operations to manage the routing elements of the FPGA.*

Operation[a]	**Description**
trace(source)	Provides the set of pass transistors currently used by the line with origin *source*.
neighbours(source)	Provides the set of pass transistors that can connect *source* to adjacent segments.
trace(source, destination)	Provides the set of pass transistors currently used to route *source* to *destination*.
route(source, destination)	Provides the set of pass transistors that could be used to connect *source* with *destination*.

[a] *source* and *destination* refer to elements along a routing line, either CB input/output pins or pass transistors.

These operations will be used throughout the present thesis to ease the task of specifying how to manage the configuration memory of the FPGA to emulate

the behaviour of the system in the presence of the desired fault. Obviously, the actual implementation of these operations will depend on the architecture of the selected FPGA.

3.5 Conclusions

Model-based fault injection techniques can be applied in the first stages of the development cycle, thus reducing the cost of fixing any error on the design. However, the time required to simulate large and complex models restricts the applicability of these techniques.

FPGAs have been proposed as a means to accelerate model-based fault injection experiments by implementing the model of the system. The use of FPGAs with fault injection purposes is known as *fault emulation*, and joins the benefits of prototype- (fast execution) and model-based (early and low cost validation) fault injection techniques. Two different techniques, *Compile-* and *Run-Time reconfiguration*, have evolved from the original *serial fault emulation* to improve the execution time of fault injection experiments.

Compile-Time Reconfiguration presents a negligible reconfiguration time, although the model instrumentation may increase the size of the model beyond the capabilities of the selected FPGA. In this case, several partial instrumentations are mandatory to cover the whole set of desired fault experiments, thus resulting in long synthesis and implementation processes.

Run-Time Reconfiguration only requires a single implementation of the system's model. Even though the programmable device must be reconfigured several times per experiment, the temporal cost of the reconfiguration processes is greatly counterbalanced by the time saved by implementing the model once.

Since one of the main goals of this thesis is to accelerate model-based fault injection experiments for a wide set of hardware fault models considered representative of new deep-submicron manufactured systems, Run-Time Reconfiguration has been selected as the best suitable methodology in order to obtain the better speed-up ratio when injecting faults into these kind of systems.

That particular fault injection technique is closely related to the architecture of the considered programmable device. Hence, to make this study as comprehensive as possible, the generic FPGA architecture proposed in Section 3.4 will be used to develop new methodologies for the injection of all the considered hardware fault models. This generic approach could be easily adapted later to the particular architecture of any available FPGA.

Chapter 4

New Approaches for Transient and Permanent Faults Emulation

Deep-submicron technologies have brought into scene new and more complex fault models, such as pulse, stuck-opens, indeterminations, delays, shorts, open-lines, *and* bridgings, *apart from the classical* bit-flips *and* stuck-ats.

Up to now FPGAs have been successfully used to emulate the occurrence of the well-known bit-flips *and* stuck-ats, *but the rest of faults considered representative of new semiconductor technologies have not been address yet.*

This Chapter describes novel approaches for emulating the occurrence of the proposed fault models into the system's model by using FPGAs. Since it adopts a Run-Time Reconfiguration methodology, the reconfiguration time devoted to the injection of each fault is reported. A comparison among the different possibilities is also presented when several options are available.

4.1 Introduction

On the one hand, transient faults appear during the operation of the circuit for a short period of time, usually as a result from the interference or interaction of the circuitry with its physical environment. On the other hand, permanent faults are due to irreversible physical defects in the circuit as a result of the manufacturing process or the normal operation of the system.

Although FPGAs have been successfully used to emulate the occurrence of *bit-flips* and *stuck-ats*, Sections 2.2 and 2.3 exposed the necessity of taking into account new and more complex fault models at logic and RT levels when assessing the dependability of deep submicron manufactured systems, like the

transient *pulses*, *indeterminations* and *delays*, and the permanent *stuck-opens*, *indeterminations*, *delays*, *short*, *open-lines*, and *bridgings*.

Systems described at the RT level account how data is transformed as it passes from register to register. The generic configurable block presented in Section 3.4, which consists in a LUT, implementing the combinational logic of the circuit, that drives the associated FF, nicely match an RT level description. Hence, FPGAs are very well-suited for the implementation of systems described in the logic and RT levels and, as a result, the emulation of faults into these description levels by means of FPGAs seems quite promising.

Run-Time Reconfiguration, the selected approach for fault emulation, relies in the dynamic reallocation of FPGA's resources to emulate the occurrence of faults into the system's model. This is achieved by reconfiguring the FPGA, i.e. by modifying the contents of its configuration memory to change the behaviour of the configurable elements of the programmable device. Hence, operations listed in Table 3.1 and Table 3.2 are considered to modify the configuration memory of the FPGA and thus the behaviour of its configurable elements.

When dealing with transient faults the fault remains into the system until its duration expires. After that, and according to the proposed approach, the internal resources of the FPGA have to be dynamically reallocated again to restore the previous fault-free configuration of the system. However, the state of the system is kept untouched, as it could have been modified by the fault.

However, due to their nature, permanent faults do not expire and remain in the system forever.

Therefore, the main goal of this Chapter is to detail the proposed methodology for the injection of each considered transient [83] and permanent [84] fault model following a Run-Time Reconfiguration approach. This study will determine which are the logic elements affected by those faults, how they are mapped to FPGAs resources, and how these resources can be reconfigured to emulate the behaviour of the faulty system.

Although the considered methodology is flexible enough to offer a range of different possibilities, like emulating transient *shorts*, this work has been focused on emulating the occurrence of the fault models considered representative of deep-submicron technologies as defined by the representativeness study [4].

All the identified approaches are analysed in terms of their applicability and the time devoted to the FPGA reconfiguration. The emulation of *bit-flips* is presented in Section 4.2, and *stuck-ats* emulation appears in Section 4.3. Sections 4.4, 4.5, and 4.6, detail the emulation of *pulses*, *stuck-opens*, and *indeterminations*, respectively, which are all based on the previously presented

approaches for the emulation of *stuck-ats*. After that, the injection of *delays* is described in Section 4.7. The emulation of faults based on rerouting the circuit implemented on the FPGA, like *shorts*, *open-lines*, and *bridgings*, is presented in Sections 4.8, 4.9, and 4.10, respectively. Finally, Section 4.11 summarises the results of this analysis and draws the main conclusions of this work.

4.2 Emulation of *bit-flips*

Bit-flip is one of the most used transient fault models which consists in inverting the current logic state of a memory cell. It has no associated duration and, as a result, there is no need to delete the fault. The affected bit holds the faulty logic value until it is rewritten by the normal operation of the system.

The occurrence of a *bit-flip* into a system's model will affect those ports, signals, and variables, that are used to implement the sequential logic (memory elements) of the system. So, these are the injection points that are taken into account when injecting *bit-flips* into the system's model.

However, once the model is synthesised and implemented onto an FPGA, it is necessary to locate the internal components those injection points have been mapped to in order to inject the very same faults but using an FPGA.

Figure 4.1 depicts a hardware component with some sequential elements described in VHDL, and how their mapping to an FPGA once synthesised.

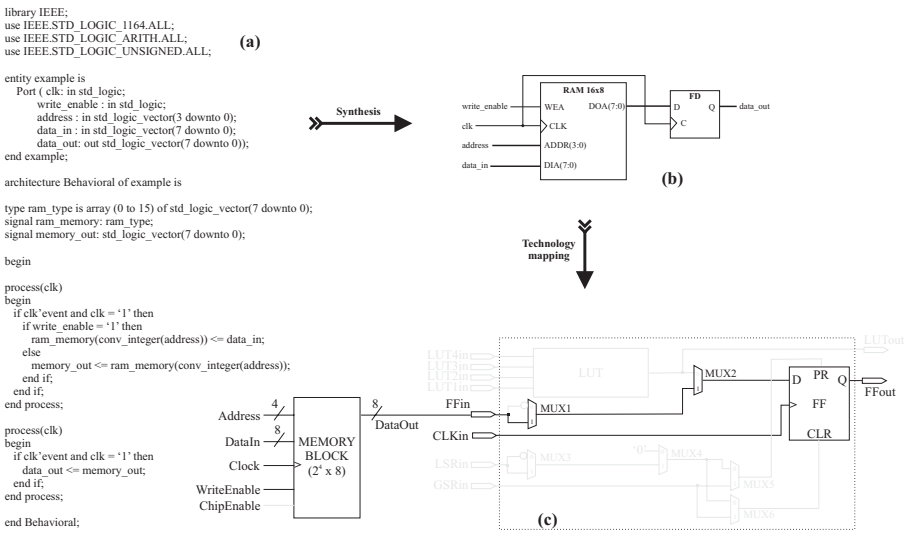

Figure 4.1: *Example of the implementation of a sequential circuit by means of an FPGA. The model of the sequential circuit (a) is synthesised to extract the described logic (b). This logic is then mapped to the internal elements of the FPGA (c).*

According to Figure 4.1a, signals *ram_ memory* and *memory_ out*, describe a 16x8 RAM memory and a 8-bit flip-flop respectively. So these are the elements that could be affected by *bit-flips* during the model's simulation.

Considering the technology mapping shown in Figure 4.1c, the implementation of this model onto and FPGA will map these elements to a memory block and a FF in a CB. Thus, these are the injection points that can be considered when emulating the occurrence of *bit-flips* and, in general, any kind of faults targeting the sequential logic of the system.

First of all, we will consider the occurrence of a *bit-flip* in a memory block, since it is the simplest of the approaches and could exemplify the procedure to be followed for the rest of the considered faults. After that, the emulation of this kind of faults in FFs will be addressed.

4.2.1 Injecting *bit-flips* into memory blocks

This fault will affect one of the bits of the selected memory block.

Its emulation is as simple as obtaining the current logic state of the targeted memory cell and writing the opposite one. The following steps, depicted in Figure 4.2, detail that process.

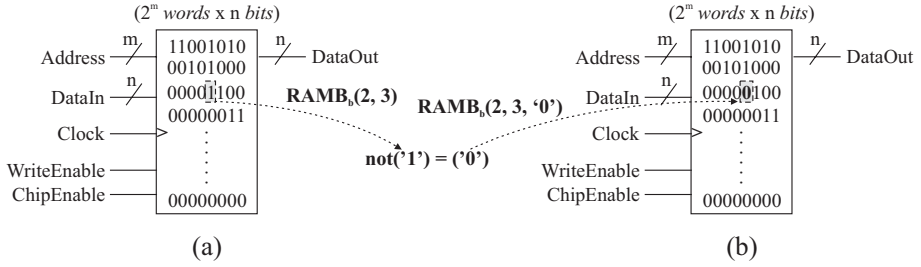

Figure 4.2: *Emulating the occurrence of a* bit-flip *into a memory block. The content of the fault-free memory cell (a) is reversed by the fault (b).*

a) The first step consists in reading the current value of that bit to determine which is the logic state that must be induced in the memory cell. It can be obtained by reading the proper bit from the configuration memory of the FPGA (value = RAMB$_b$(word, bit)).

b) The new expected logic value of the affected memory cell can be easily drawn from this information (from `'0'` to `'1'` or vice versa). This new value is then written into the configuration memory of the FPGA to modify the contents of the memory block (RAMB$_b$(word, bit, $\overline{value}$)).

In that way, the system now behaves as if a *bit-flip* fault had occurred.

Considering that reading and writing a bit from the configuration memory of the FPGA takes one time unit (T), the time required to reconfigure the FPGA to inject a *bit-flip* into a memory block is computed by Equation 4.1.

$$\begin{aligned} (\textit{read the state of the memory cell}) \quad & T+ \\ (\textit{invert the state of the memory cell}) \quad & T = \\ \textbf{(total)} \quad & \textbf{2T} \end{aligned} \tag{4.1}$$

4.2.2 Injecting *bit-flips* into FFs

In general, the occurrence of faults is not synchronised with the execution of the system. This causes that the only way to asynchronously change the state of a FF is to trigger the signals that control its set and reset logic.

As can be seen in Figure 3.13, the set/reset logic of FFs is controlled by the Global Set Reset (GSRin) line, which can be pulsed by sending the proper command (represented by the function *TriggerGSRLine()*) to the FPGA.

Thus, the methodology proposed in [59] is based on the use of the GSRin line. This common approach is next described to better understand this injection process and show its main drawbacks. Afterwards, a new methodology that improves the speed-up obtained by this technique is presented [85].

4.2.2.1 Using the Global Set Reset (GSRin) line

The GSRin line, as its name states, is a *global* line that drives the set/reset logic of *all* the FFs of the FPGA. Hence, pulsing that line will cause the activation of the set or reset (depending on the configuration of the associated multiplexers) of all the FFs of the system.

The use of the GSRin line for emulating the occurrence of *bit-flips* into FFs involves the following steps, which are also illustrated in Figure 4.3.

a) The state of *all* the FFs of the system must be read ($\text{FF}_f()$). This information is not only necessary to invert the current logic value of the FF affected by the fault, but also to keep the current logic value of the rest of FFs.

b) According to this, the multiplexers that manage the set/reset of the FF (MUX5 and MUX6, cf. Figure 3.13) must be properly reconfigured.

   In case that a FF must hold its current low logic level, the suitable operations for the GSRin line to drive the clear input of the FF are

MUX5$_i$(‘0’) and MUX6$_i$(‘1’). If the FF must keep a high logic level, its set input will be fed by the GSRin line by executing MUX5$_i$(‘1’) and MUX6$_i$(‘0’).

When considering the inversion of the FF's current low logic state, the appropriate operations to connect the FF's set input to the GSRin line are MUX5$_i$(‘1’) and MUX6$_i$(‘0’). On the other hand, inverting a high logic state requires the execution of MUX5$_i$(‘0’) and MUX6$_i$(‘1’) for the GSRin line to drive the FF's clear input.

c) Now, the GSRin line can be pulsed (TriggerGSRLine()) to reverse the current state of the FF targeted by the fault.

d) Once the fault has been injected into the system, those multiplexers that were reconfigured to allow for the fault injection must be restored to their previous configuration (MUX5$_i$(‘0’/‘1’) and MUX6$_i$(‘0’/‘1’)).

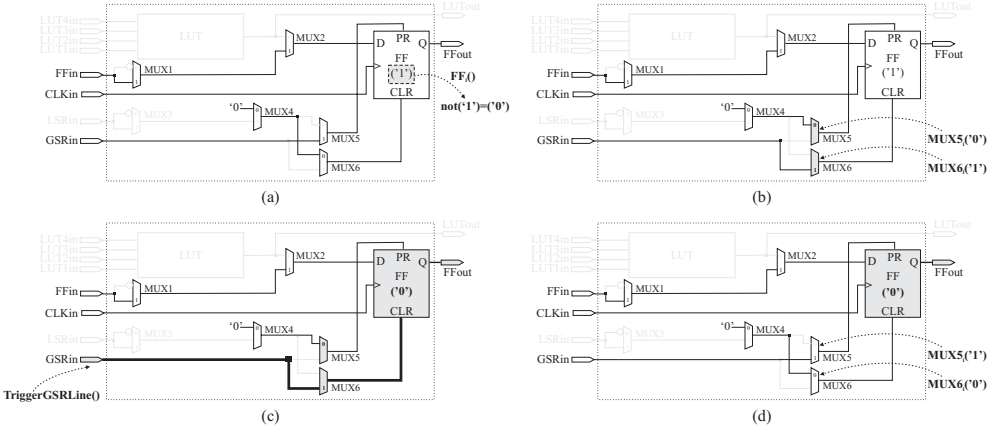

Figure 4.3: *Emulating the occurrence of a* bit-flip *into a FF by using the GSRin line. The content of the fault-free FF is read (a) to determine how to configure the multiplexers that control the set/reset logic of the FF (b). Pulsing the GSRin line (c) flips the logic state of the FF and, after that, the multiplexers are reconfigured to their previous state (d).*

The main problem this common approach has to face is inherent to the use of the GSRin line to activate the set/reset of the affected FF. This *global* line drives all the FFs of the device, causing that the state of all the FFs must be read and all the MUX5 and MUX6 multiplexers must be reconfigured twice (to inject the fault and restore its previous configuration).

Let us assume that reading/writing a bit from/to the configuration memory of the FPGA takes one time unit (T). According to this, and considering

the steps involved in applying the GSRin-based methodology, the FPGA's reconfiguration time when injecting a *bit-flip* into a system comprising n FFs is computed by Equation 4.2.

$$\begin{aligned} (read\,the\,state\,of\,all\,the\,FFs) & \quad nT+ \\ (configure\,all\,the\,multiplexers\,MUX5\,and\,MUX6) & \quad 2nT+ \\ (pulse\,the\,GSRin\,line) & \quad 0+ \\ (restore\,the\,previous\,state\,of\,all\,the\,multiplexers) & \quad 2nT = \\ \textbf{(total)} & \quad \mathbf{5nT} \end{aligned} \tag{4.2}$$

This is the worst case function, since it considers that *all* the multiplexers have to be reconfigured to inject the fault into the system. Assuming that m is the percentage of multiplexers that are already properly configured to inject the fault, the time devoted to the FPGA reconfiguration is computed by Equation 4.3.

$$\begin{aligned} (read\,the\,state\,of\,all\,the\,FFs) & \quad nT+ \\ (configure\,m\%\,of\,multiplexers) & \quad 2n\frac{m}{100}T+ \\ (pulse\,the\,GSRin\,line) & \quad 0+ \\ (restore\,the\,state\,of\,m\%\,of\,multiplexers) & \quad 2n\frac{m}{100}T = \\ \textbf{(total)} & \quad (\mathbf{n + 4n\frac{m}{100}})\mathbf{T} \end{aligned} \tag{4.3}$$

4.2.2.2 Using the Local Set Reset (LSRin) line

As one of the main goals of this work is to accelerate, as much as possible, the execution of model-based fault injection experiments, we have devised a novel approach for injecting *bit-flips* into FFs that has the merit of reducing the time devoted to the injection of the fault. This approach is also based on activating the set/reset signals of the affected FF but, instead of using the GSRin line for that purpose, the Local Set Reset line (LSRin) is proposed as an alternative.

As its name states, the LSRin line is *local* to the associated FF. This means that the activation of that line only triggers the set or reset of the FF it drives and the rest of the circuit remains unaffected. It is mainly used to implement elements, such as counters or registers, that must be individually set to a particular state at some moment (when reaching a certain count, for example).

The main problem with the LSRin line is that it cannot be directly pulsed as in the case of the GSRin line. Therefore, as it must be pulsed by the normal operation of the circuit, it has not been considered yet for the emulation of *bit-flips* into FFs. Nevertheless, we have contrived a tricky use of FPGA's internal resources that enables the activation of the LSRin line by following the steps depicted in Figure 4.4.

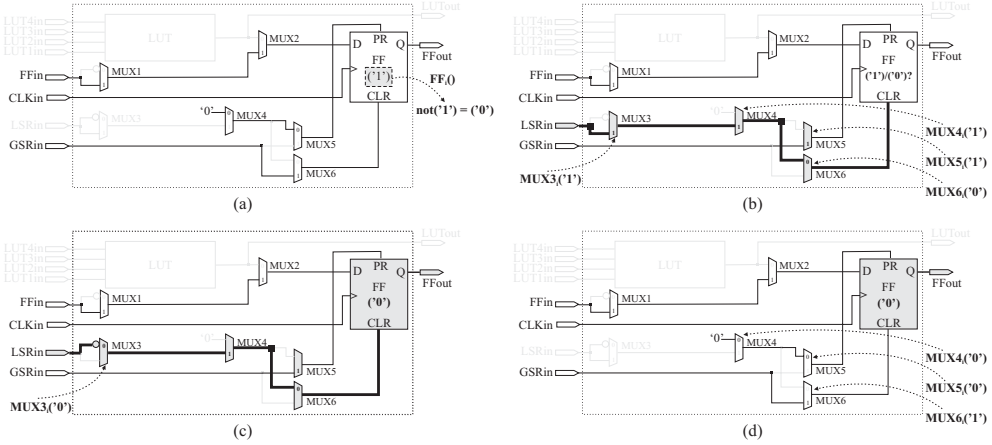

Figure 4.4: *Emulating the occurrence of a* bit-flip *into a FF by using the LSRin line. The state of the fault-free FF is read (a) to determine how to configure the multiplexers that control the set/reset logic of the FF (b). The LSRin line is also connected to the set/reset logic. The LSRin line is pulsed (c) by inverting its incoming logic value to flip the state of the FF and, after that, all the involved multiplexers are reconfigured to their previous state (d).*

a) The state of *only* the targeted FF must be read ($\mathrm{FF}_f()$). This clearly outperforms the GSRin-based approach that required reading the state of *all* the FFs of the system.

b) According to this information, the multiplexers that control the set/reset logic of the FF (MUX5 and MUX6) must be properly reconfigured.

   If the current state of the FF must be changed to a low logic state, $\mathrm{MUX5}_i(\texttt{'1'})$ and $\mathrm{MUX6}_i(\texttt{'0'})$ are the proper operations to be performed. In case that the FF's final state should be a high logic state, its set input line will be fed by the LSRin line by executing $\mathrm{MUX5}_i(\texttt{'0'})$ and $\mathrm{MUX6}_i(\texttt{'1'})$.

   Again, this step only affects the multiplexers associated to the targeted FF and not the rest of multiplexers of the system.

The multiplexer MUX4 must also be configured to make the LSRin line drive the set/reset lines of the FF ($MUX4_i$(‘1’)).

c) Now, the LSRin line must be pulsed to reverse the current state of the selected FF. Even though that line cannot be externally pulsed, the procedure to achieve that goal follows.

Whether the LSRin line is being used or not in the system, it is not possible to ascertain the logic state that line is holding at the fault injection time. If the line is in use, it can be driven by some combinational logic and, therefore, its current state cannot be guessed. If the line is not used, it is usually bound to a logic ‘1’ or logic ‘0’ depending on the particular architecture of each FPGA.

Nonetheless, the multiplexer that is in charge of inverting the incoming value of that signal (MUX3) can be used to cause a pulse in the line. No matter what is the current logic state of the line, this multiplexer can invert the incoming value of the LSRin line ($MUX3_i$(‘0’)) and, after that, it can be configured to drive the original logic value of the signal ($MUX3_i$(‘1’)). This sequence of operations will cause a pulse in the LSRin line that will assure the activation of the set/reset logic of the FF whatever the logic state of the LSRin line is.

d) In the last stage, all the multiplexers that have been used in this process must be restored to their previous configuration ($MUX3_i$(‘0’/‘1’), $MUX4_i$(‘0’/‘1’), $MUX5_i$(‘0’/‘1’) and $MUX6_i$(‘0’/‘1’)).

Equation 4.4 computes the FPGA's reconfiguration time required to inject a *bit-flip* using the LSRin line, assuming that reading/writing a bit from/to the configuration memory of the FPGA takes one time unit (T), and that all the multiplexers involved in the process must be reconfigured to inject the fault.

$$\begin{aligned} (read\ the\ state\ of\ one\ FF) &\quad T+ \\ (configure\ the\ multiplexers\ MUX5\ MUX6\ and\ MUX4) &\quad 3T+ \\ (pulse\ the\ LSRin\ line\ by\ reconfiguring\ MUX3) &\quad T+ \\ (restore\ the\ previous\ state\ of\ the\ multiplexers\ involved) &\quad 4T = \\ \textbf{(total)} &\quad \textbf{9T} \end{aligned} \tag{4.4}$$

Equation 4.5 presents the reconfiguration time required to emulate that fault in case that the m% of multiplexers are not properly configured yet.

$$
\begin{aligned}
(read\ the\ state\ of\ one\ FF) \quad & T+ \\
(configure\ m\%\ of\ multiplexers) \quad & 3\frac{m}{100}T+ \\
(pulse\ the\ LSRin\ line\ by\ reconfiguring\ MUX3) \quad & T+ \\
(restore\ previous\ state\ of\ involved\ multiplexers) \quad & (1+3\frac{m}{100})T = \\
\textbf{(total)} \quad & \mathbf{(3+6\frac{m}{100})T}
\end{aligned}
\tag{4.5}
$$

4.2.2.3 Discussion

As can be clearly seen from Equations 4.4 and 4.5, the LSRin-based approach does not depend on the number of FFs of the considered system.

Figure 4.5 shows the temporal cost of the reconfiguration associated to the use of these two approaches in terms of number of FFs (n) and the percentage of multiplexers that have to be reconfigured (m).

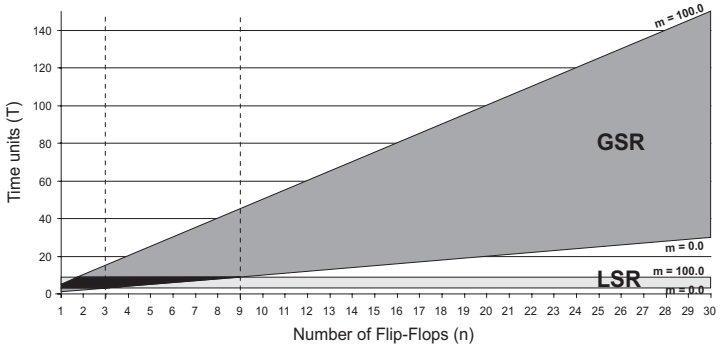

Figure 4.5: *Cost in time units of injecting a* bit-flip *into a FF by using the GSR and the LSR approaches. The cost of using the GSRin line, depending on the percentage of multiplexers that must be reconfigured (*m*), appears in dark grey. The temporal cost of using the LSR approach is depicted in light grey. The black coloured area indicates the range where both approaches could be indistinctly used.*

The common GSR approach may have some benefits when considering a system comprising a maximum of 3 FFs, whereas the use of the LSRin line is the best approach for any system with more than 9 FFs. Taking into account Equations 4.4 and 4.5, and depending on the parameters n and m, the speed-up ratio attainable by the conventional approach ranges between $\frac{n}{3}$ ($m = 0\%$) and $\frac{5n}{9}$ ($m = 100\%$).

Hence this novel approach greatly accelerates the injection of the *bit-flip* faults into modern complex systems.

4.2.3 Summary

Bit-flips may affect any sequential element of the system, which are usually implemented as either memory cells of flip-flops.

A novel methodology has been presented to emulate the occurrence of *bit-flips* into the flip-flops of the system, which speeds-up the conventional approach that has been followed up to now.

The emulation of *bit-flips* by means of the different considered approaches is summarised following a C-like pseudo-code in Table 4.1.

Table 4.1: *Pseudo-code for injecting a* bit-flip *into the system.*

Fault injection

```
switch (target) {
   case MEMORY_BLOCK:
     stateBit = RAMB_j(word, bit);
     RAMB_j(word, bit, not stateBit);
     break;
   case FF:
     stateFF_j = FF_j();
     for (i = 3; i < 7; i++) stateMUX[i]_j = MUX[i]_j();
     if (existsLSRin) {
       MUX[5]_j(stateFF_j);
       MUX[6]_j(not stateFF_j);
       MUX[4]_j('1');
       MUX[3]_j(not stateMUX[3]_j);
       for (i = 3; i < 7; i++) MUX[i]_j(stateMUX[i]_j);
     } else {
       MUX[5]_j(not stateFF_j);
       MUX[6]_j(stateFF_j);
       MUX[4]_j('0');
       for (i = 0; i < #FFS; i++) {
         if (i != j) {
           stateFF_i = FF_i();
           for (k = 4; k < 7; k++) stateMUX[k]_i = MUX[k]_i();
           MUX[5]_i(stateFF_i);
           MUX[6]_i(not stateFF_i);
           MUX[4]_i('0');
         }
       }
       TriggerGSRLine();
       for (k = 0; k < #FFS; k++)
         for (i = 4; i < 7; i++) MUX[i]_k(stateMUX[i]_k);
       }
     }
     break;
}
```

4.3 Emulation of stuck-ats

The *stuck-at* is one of the most popular used permanent fault models, in which the logic state of the targeted element is bound to a low- (*stuck-at-0*) or high-logic level (*stuck-at-1*) regardless the evolution of its inputs.

This fault may affect both the combinational and sequential logic of the system's model. Then, LUTs (combinational logic) and FFs (sequential logic) are the FPGA's internal resources that must be reconfigured when emulating the occurrence of *stuck-ats* in the system.

The following Sections will deal with the emulation of *stuck-ats* affecting these configurable elements.

4.3.1 Injecting *stuck-ats* into combinational logic

As no real combinational logic is usually available in the architecture of current FPGAs, combinational functions are implemented by means of LUTs.

The synthesis process extracts the combinational logic described by the system's model, which is optimised to obtain a reduced combinational circuit with the same functionally. This reduced circuit is divided into 4-variable functions to fit the 4-input LUTs present in the generic FPGA architecture described in Section 3.4. This whole process is shown in Figure 4.6.

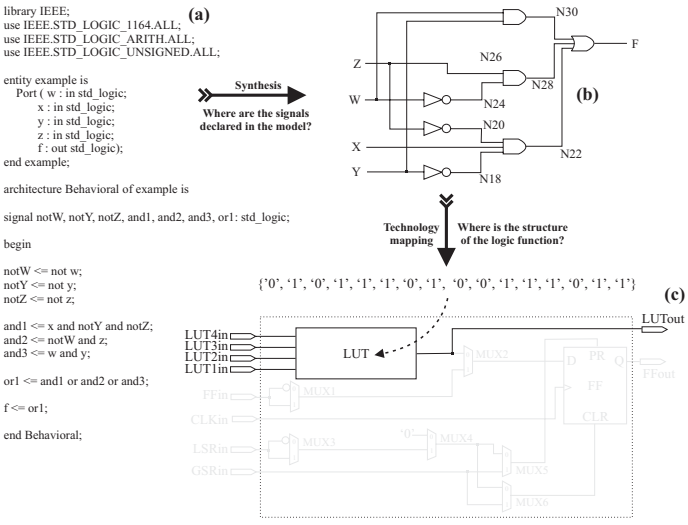

Figure 4.6: *Example of the implementation of a combinational logic function by means of a 4-input LUT. The model that describes the combinational function (a) is synthesised to obtain a structural representation of the circuit (b). This circuit is divided into 4-variable functions which are mapped to the LUTs of the FPGA (c).*

The occurrence of *stuck-ats* into the combinational logic of the system's model will affect those ports, signals, and variables, implementing that logic. So, according to Figure 4.6a, ports *w*, *x*, *y*, *z*, and *f*, and signals *notW*, *notY*, *notZ*, *and1*, *and2*, *and3*, and *or1*, are eligible for being targeted by these faults.

However, LUTs have been usually considered as black boxes that implement the combinational logic of the system. For that reason, *stuck-at* faults were only injected at the input and output ports of these black boxes (*w*, *x*, *y*, *z*, and *f* in Figure 4.6b), which was known as Line Stuck-At (LSA).

Clearly, this approach poses a representativeness problem, since it does not really cover all the possibilities for the realistic occurrence of these faults in the logic structure. For instance, the insertion of *stuck-at* faults at the inputs and output of the LUT depicted in Figure 4.6 will not consider *stuck-at* faults that may occur into the logic structure of the combinational circuit it represents (NOT gates N24, N20, and N18, and AND gates N30, N28, and N22).

To cope with this problem, a new fault model, named *Combination Stuck-At* (CSA), was suggested in [86] [87]. It consists in extracting a structural representation of the combinational circuit implemented by the LUT to consider the possible occurrence of faults in the logic gates and lines of that circuit.

We have considered this approach, as shown in Figure 4.7, to cover the occurrence of *stuck-ats* and, in general, of any kind of faults into the combinational logic of the system implemented as a LUT.

a) The LUT contents represent the truth table of the combinational function implemented by this element ($LUT_l()$). A structural representation of that circuit can be extracted from this information by using, for instance, the Quine-McCluskey algorithm [88] [89].

b) The fault may affect any logic element, signal, input and output of the extracted circuit. Once that the fault injection point has been selected, the structural representation of the circuit is modified to take into account the occurrence of the fault. Since dealing with *stuck-ats*, they can be emulated by considering that the affected line is now being driven a low (*stuck-at-0*) or high (*stuck-at-1*) logic level.

   The new truth table of the faulty circuit is then computed (faultyLUT = new LUT_l(fault location, stuck-at)).

c) Now, the LUT have to be reconfigured to emulate the behaviour of the system in the presence of the fault. This is accomplished by modifying the contents of the proper bits of the configuration memory to match the new computed faulty truth table (LUT_l(faultyLUT)).

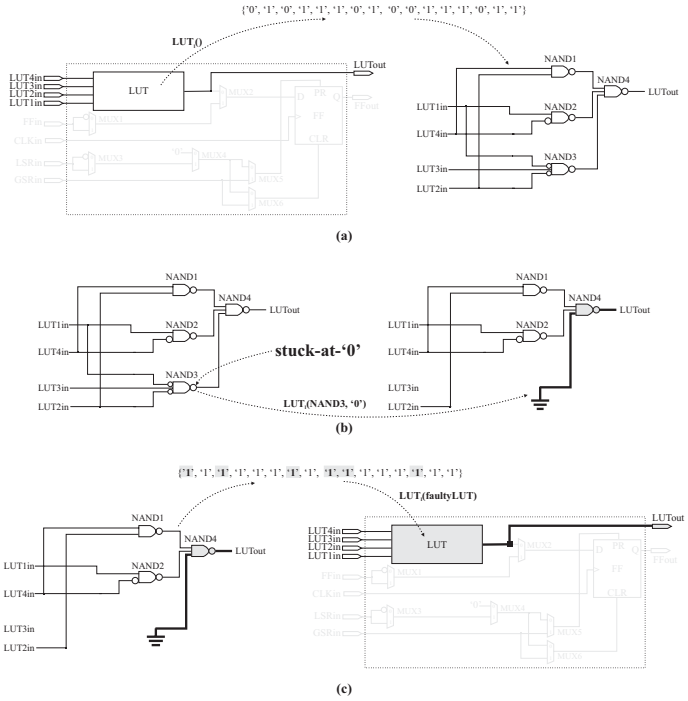

Figure 4.7: *Emulating the occurrence of a* stuck-at-0 *into combinational logic implemented as a LUT. A structural representation of the combinational circuit is extracted from the LUT's contents (a). The output of the NAND3 gate is affected by the* stuck-at-0 *(b) and the new truth table of the faulty circuit replaces LUT's contents (c) to emulate the occurrence of the fault.*

This approach emulates the occurrence of *stuck-ats* in the system model under study by modifying the contents of the LUT where the targeted combinational logic maps to. As LUTs are usually implemented as a 16x1 bit memory, and considering that writing a bit into the configuration memory of the FPGA takes one time unit (T), the time devoted to reconfigure the FPGA can be computed by Equation 4.6.

$$(configure\ the\ LUT's\ contents\ to\ inject\ the\ fault) \qquad 16T = \qquad (4.6)$$
$$\textbf{(total)} \qquad \textbf{16T}$$

This Equation assumes that all the bits of the LUT must be reconfigured. Taking into account that usually this is not the case and only the k% of these bits have to be reconfigured, the FPGA's reconfiguration time is computed by Equation 4.7.

$$(configure\ the\ k\%\ of\ LUT's\ bits\ to\ inject\ the\ fault) \qquad 16\frac{k}{100}T = \quad (4.7)$$

$$\textbf{(total)} \qquad \mathbf{16\frac{k}{100}T}$$

4.3.2 Injecting *stuck-ats* into sequential logic

The sequential logic of the system is usually implemented by means of FFs.

The occurrence of a *stuck-at* may cause the FF's current logic state to change. However, as previously explained, the only way of asynchronously modifying the state of a FF is via its set/reset signals. Consequently, the same procedures presented in Section 4.2.2 to invert the current state of the FF can be used now for the emulation of this fault.

In case that the FF already holds the desired final logic state, this step is obviously needless.

As dealing with a permanent fault, once the FF presents the desired logic state, it is necessary to keep it constant. Three different approaches to fix the logic state of the FF, based on the use of the FF's LSRin line, the LUT associated to the FF, and the FF's clock signal, are next described.

4.3.2.1 Using the unused Local Set/Reset (LSRin) line

One of the possibilities for keeping constant the current logic state of a FF is causing a continuous set (*stuck-at-1*) or reset (*stuck-at-0*). Evidently, this cannot be attained by using the GSRin signal, since it will cause all the FFs of the system to set/reset uninterruptedly. The only sensible possibility is using the LSRin signal to set or reset the affected FF.

As previously reported, unconnected CB's inputs are usually bound to a high- or low-logic level depending on the particular architecture of each FPGA. Thus, a very similar approach than that presented at Section 4.2.2 may be used.

However, if the LSRin line is being currently used by the system, there is no way to assure that it will continuously set/reset the FF. The normal operation of the system may modify the current logic state of the line at anytime.

This approach, which Figure 4.8 exemplifies, is thus applicable only when that particular LSRin line is not being used by the system.

a) Firstly, it is necessary to check whether the LSRin line is actually unused.

   A simple possibility is checking the multiplexer that enables the LSRin line to drive the set/reset logic of the FF ($MUX4_i()$). If its control bit is set to `'0'` the LSRin line will surely be unused.

A thorough examination can check the pass transistor that connects the routing logic to the LSRin line (PT_{LSRin_t}). The LSRin line will also be unused if this pass transistor is off.

In case that the line is being used, another procedure should be considered for the injection of the fault.

b) In order to fix, and at the same time change, the current logic state of the faulty FF, the multiplexers that control its set/reset logic (MUX5 and MUX6) must be properly reconfigured.

In case the FF must hold a low logic state, $MUX5_i$(`'1'`) and $MUX6_i$(`'0'`) are the proper operations to be performed. If it must keep a high logic state, the LSRin line will feed the FF's set signal when $MUX5_i$(`'0'`) and $MUX6_i$(`'1'`).

The multiplexer MUX4 must also be configured to make the LSRin line drive the set/reset lines of the FF ($MUX4_i$(`'1'`)).

Now, the multiplexer that is in charge of inverting the incoming value of the LSRin signal (MUX3) can be used to properly treat this value.

If the FPGA's architecture provides a low logic level to the unused CB's inputs, this multiplexer can invert the incoming value of the LSRin line ($MUX3_i$(`'0'`)) to activate the set/reset of the FF. Otherwise, it can be configured to drive the high logic value of the signal ($MUX3_i$(`'1'`)).

In this way, the set/reset logic of the FF is continuously activated, keeping its current logic state fixed to the desired value.

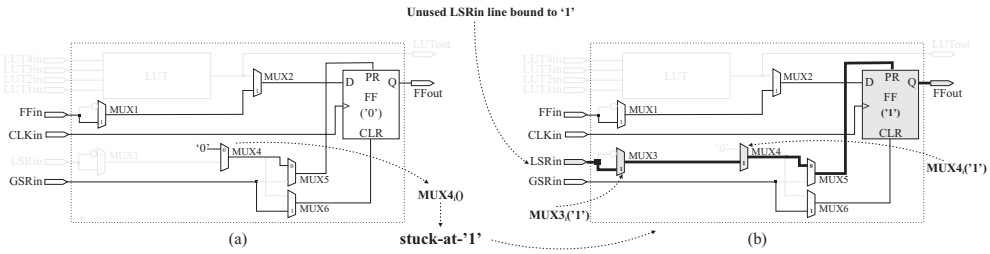

Figure 4.8: *Emulating the occurrence of a* stuck-at-1 *into sequential logic by using the LSRin signal. The configuration of the system is checked to confirm that this approach can be used (a). The multiplexers that control the set/reset logic of the FF are reconfigured for the LSRin line to feed the reset signal of the affected FF (b) thus emulating the occurrence of the fault.*

The time units (T) required to reconfigure the FPGA to emulate the occurrence of a *stuck-at* that fixes the FF's logic state by using this approach

can be determined by Equation 4.8. T stands for the the time required to read or write one bit of the FPGA's configuration memory, whereas m represents the percentage of multiplexers that must be reconfigured to keep activated the set/reset of the FF.

$$\begin{aligned} (configure\ the\ m\%\ of\ multiplexers) \quad & (1 + 2\frac{m}{100})T + \\ (reconfigure\ MUX3) \quad & T = \\ \mathbf{(total)} \quad & \mathbf{(2 + 2\frac{m}{100})T} \end{aligned} \tag{4.8}$$

4.3.2.2 Using the unused LUT associated to the FF

Another possibility to fix the logic state of the affected FF is to continuously feed it with the required value. In this way, the normal operation of the system will rewrite the same value the FF holds.

This can be achieved, for instance, by configuring the LUT present at the same CB as the FF to provide the desired logic value permanently.

Consequently, the approach next described can only be applied, as depicted in Figure 4.9, when the required LUT is not being used by the system.

a) This first step consists in determining whether the LUT present at the same CB as the FF affected by the fault can be used to inject the fault. In case that the LUT is not being used at all or that it is already feeding the affected FF and *only* that FF, this LUT is eligible to drive a constant value to that FF.

   A simple option is to check the pass transistor that connects the output of the LUT to the rest of the circuit ($\mathrm{PT}_{LUTout_t}()$). If that pass transistor is off the LUT is isolated from the rest of the circuit or, at most, connected to the input of the proper FF. If that LUT is already being used, it will be necessary to consider a different approach for injecting the *stuck-at*.

b) The state of the targeted FF is read ($\mathrm{FF}_f()$) to determine if it matches the one caused by the fault.

   If this is not the case, the very same approach presented in Section 4.2.2 must be followed to asynchronously invert the current state of the FF.

   Now, that the FF already holds the desired final value, it is necessary to continuously provide that same value to the FF's input.

   The connection between the LUT's output and the FF's input is performed by reconfiguring the proper multiplexer ($\mathrm{MUX2}_i(\texttt{'0'})$).

Finally, the truth table of the circuit that provides a constant '0' or '1' (faultyLUT = new LUT_l(LUT's output, stuck-at)) is computed and the contents of the LUT are reconfigured according to this information (LUT_l(faultyLUT)).

The normal operation of the system will not change the final logic state induced by the fault.

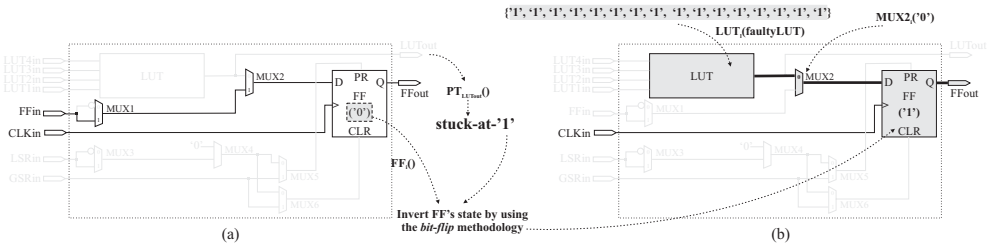

Figure 4.9: *Emulating the occurrence of a* stuck-at-1 *into sequential logic by using the LUT associated to the affected FF. The applicability of this approach is checked (a). The FF's current state must be changed by following the* bit-flip *emulation approach to match the logic state caused by the fault. The LUT's are reconfigured to provide the suitable constant value and its output is connected to the FF's input (b).*

According to this approach, the time units (T) required to reconfigure the FPGA to emulate the occurrence of this fault can be carried out by Equation 4.9. T stands for the time required read/write a bit from/to the FPGA's configuration memory, whereas m represents the percentage of multiplexers that have to be reconfigured to invert the logic state of the FF, and k is the percentage of bits that must be modified for the LUT to provide the desired constant value. As some of the steps might not be taken, the minimum and maximum reconfiguration times are computed.

$$
\begin{aligned}
(read\ the\ FF's\ current\ state) &\quad T+ \\
(invert\ the\ FF's\ state,\ if\ needed) &\quad [0,(2+6\frac{m}{100})T]+ \\
(configure\ MUX2,\ if\ needed) &\quad [0,T]+ \\
(reconfigure\ the\ LUT's\ contents) &\quad 16\frac{k}{100}T = \\
\textbf{(minimum total)} &\quad (\mathbf{1+16\frac{k}{100}})\mathbf{T} \\
\textbf{(maximum total)} &\quad (\mathbf{4+6\frac{m}{100}+16\frac{k}{100}})\mathbf{T}
\end{aligned}
\tag{4.9}
$$

4.3.2.3 Using the clock signal (CLKin) of the FF

The last of the proposed approaches to fix the logic state induced by the *stuck-at* consists in neutralising the events that cause the FF to change its current state.

As a synchronous element, a D-type FF changes its state by the one provided by its input line at each rising or falling clock edge. However, if no edge is provided by the clock signal, the FF will keep its value indefinitely.

This goal could be attained by turning off the pass transistor that connects the targeted FF to the clock network.

The following steps show how to achieve this aim (see Figure 4.10).

a) First of all, the current state of the affected FF is read ($\mathrm{FF}_f()$).

   If it do not match the final logic state caused by the *stuck-at*, the approach presented in Section 4.2.2 has to be followed to asynchronously invert the FF's state.

b) Now, the pass transistor that routes the clock signal to the FF is turned off to prevent any clock edge from arriving to the FF ($\mathrm{PT}_{CLKin_t}(\mathrm{off})$).

   Hence, the FF will keep the final logic state of the fault.

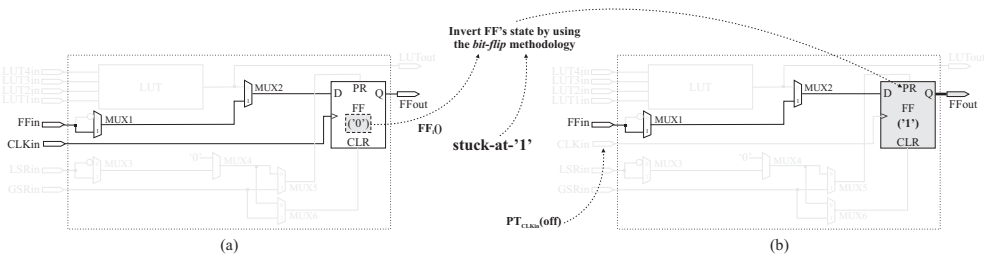

Figure 4.10: *Emulating the occurrence of a* stuck-at-1 *into sequential logic by using the clock input signal associated to the targeted FF. The FF's current state is checked to determine whether it must be reversed by following the* bit-flip *emulation approach to match the logic state caused by the fault. The FF is disconnected from the clock network by turning off the suitable pass transistor, thus preventing the FF from receiving any clock edge (b).*

Equation 4.10 computes the best and worst number of time units (T) required to reconfigure the FPGA to apply this approach for the emulation of a *stuck-at*. T is the time required to access the FPGA's configuration memory for reading or writing a bit, and m represents the percentage of multiplexers that must be reconfigured to reverse the state of the FF.

$$\begin{aligned} (read\ the\ FF's\ current\ state) &\quad T+ \\ (invert\ the\ FF's\ state,\ if\ needed) &\quad [0,(2+6\frac{m}{100})T]+ \\ (disconnect\ the\ FF\ from\ the\ clock\ network) &\quad T = \\ \textbf{(minimum total)} &\quad \mathbf{2T} \\ \textbf{(maximum total)} &\quad (\mathbf{4+6\frac{m}{100}})\mathbf{T} \end{aligned} \tag{4.10}$$

4.3.2.4 Discussion

Three different approaches, using the *LSRin* line to continuously set/reset the FF, using the LUT associated to the FF to continuously drive a constant value to the FF's input, and disconnecting the *CLKin* line from the clock network to prevent the FF from receiving any clock edge, have been proposed to emulate the occurrence of *stuck-ats* into sequential logic.

As can be seen in Figure 4.11 the use of the LUT to fix the state of the FF results in the highest reconfiguration times. The best results are provided by the LSRin-based and the CLKin disconnection approaches. The use of the LSRin-based approach presents a linear distribution which is on average better than the CLKin line disconnection.

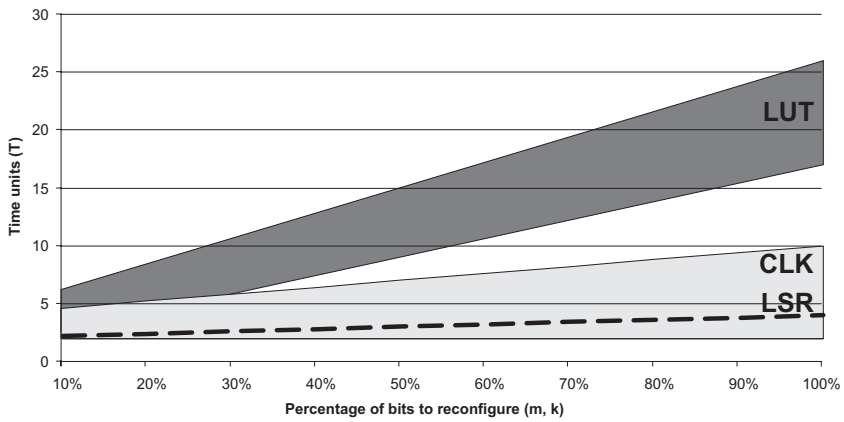

Figure 4.11: *Reconfiguration temporal cost in time units for injecting a* stuck-at *into a FF by using the LSR, LUT and CLKin approaches. The cost of using a LUT, depending on the value of* m *(percentage of bits that must be reconfigured), appears in dark grey. The temporal cost of disconnecting the CLKin line is depicted in light grey. The dashed line indicates the temporal cost of using the LSRin signal to keep fixed the FF's faulty logic state.*

Hence, the best method consists in using the LSRin line, although the CLKin one is the second option when the former is not available. The LUT-related approach incurs a long reconfiguration time and thus should be avoided whenever possible.

4.3.3 Summary

Stuck-ats are permanent faults that may affect both the sequential and combinational logic of the system.

The representativeness problem that can appear when considering the occurrence of faults into the combinational logic of the system, which is implemented by means of LUTs, have been partially solved by following the approach proposed in [86] under the name of Combination Stuck-At.

The occurrence of *stuck-ats* into sequential logic requires the use of the previously presented approach for asynchronously changing the state of the affected FF (see 4.2.2). Three different approaches have been proposed for keeping the FF's state constant afterwards.

The process for emulating the occurrence of *stuck-ats* by means of the different considered approaches is summarised following a C-like pseudo-code in Table 4.2.

4.4 Emulation of pulses

Pulses model the occurrence of Single Event Transients (SETs) into combinational logic, causing the inversion of its logic state (from high to low and vice versa) for the duration of the fault.

Although its effect is similar to that of the *bit-flips* into sequential logic, as dealing with combinational logic, the logic state induced by the fault is usually rewritten after the fault disappears from the system.

The combinational logic of the system model under study is implemented by means of LUTs. Thus, these are the elements that must be taken into account for reconfiguration to emulate the occurrence of *pulses*.

In first place, the emulation of *pulses* via LUTs' reconfiguration will be addressed. After that, we will present another approach for injecting these kind of faults in some particular cases.

Table 4.2: *Pseudo-code for injecting a* stuck-at *into the system.*

Fault injection

```
if (fault == stuck-at-'0') finalValue = '0';
else finalValue = '1';
switch (target) {
   case LUT:
     stateLUT_j = LUT_j();
     faultyLUT = LUT_j(location, finalValue);
     LUT_j(faultyLUT);
     break;
   case FF:
     switch (approach) {
       case LSRin:
         if ((MUX[4]_j() == '0') || (PT_LSRin_j() == off)) {
           for (i = 3; i < 7; i++) stateMUX[i]_j = MUX[i]_j();
           MUX[5]_j(not finalValue);
           MUX[6]_j(finalValue);
           MUX[4]_j('1');
           MUX[3]_j('1');
         }
         break;
       case LUT:
         if (PT_LUTout_j() == off) {
           if (FF_j() != finalValue) injectBitFlip();
           stateMUX[2]_j = MUX[2]_j();
           stateLUT_j = LUT_j();
           faultyLUT = LUT_j(output, finalValue);
           MUX[2]_j('0');
           LUT_j(faultyLUT);
         }
         break;
       case CLKin:
         if (FF_j() != finalValue) injectBitFlip();
         PT_CLKin_j(off);
         break;
     }
     break;
}
```

4.4.1 Injecting *pulses* into combinational logic implemented as a LUT

Section 4.3.1 described how to manage the LUT's contents to fix the logic value provided by an element of a combinational circuit. We have extended this approach, as shown in Figure 4.12, to cover the occurrence of *pulses* into combinational logic implemented as a LUT.

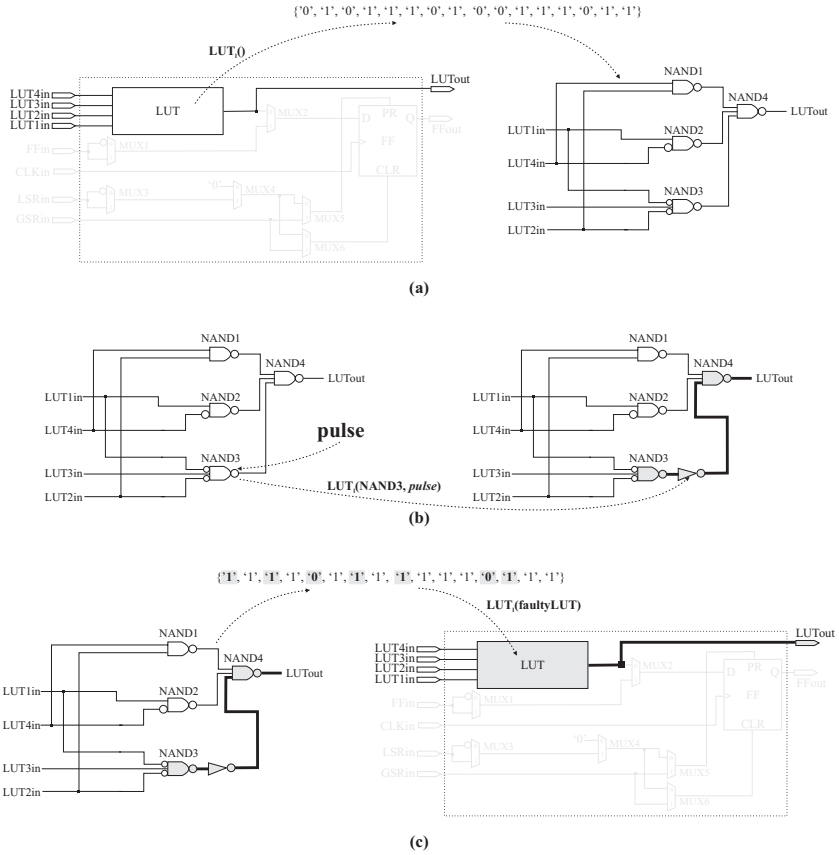

Figure 4.12: *Emulating the occurrence of a* pulse *into combinational logic implemented as a LUT. The LUT's contents are read to obtain a structural representation of the combinational circuit the LUT implements (a). The output of the NAND3 gate is affected by the* pulse *(b) and the new truth table of the faulty circuit replaces the contents of the LUT (c) to emulate the occurrence of the fault.*

The main idea consists in modifying the structural representation of the circuit implemented by the LUT to emulate the occurrence of that fault. So, a *not* gate may be inserted at the injection point to achieve this goal.

Since dealing with a transient fault, the system is again reconfigured after the fault disappears to restore the original fault-free contents of the affected LUT (LUT_l(faultFreeLUT)). In that way, the fault is deleted from the system but not its effects.

Assuming that only the k% of the LUT's bits must be reconfigured, the FPGA's global reconfiguration time for injecting and deleting the fault is computed by Equation 4.11.

$$\begin{aligned} (configure\ k\%\ of\ LUT's\ bits\ to\ inject\ the\ fault) \quad & 16\frac{k}{100}T+ \\ (restore\ the\ LUT's\ contents\ to\ delete\ the\ fault) \quad & 16\frac{k}{100}T = \\ \textbf{(total)} \quad & \mathbf{32\frac{k}{100}T} \end{aligned} \tag{4.11}$$

4.4.2 Injecting *pulses* by using inverter multiplexers

This is a particular approach that, although can only be applied to some special cases, may accelerate the emulation of *pulses* with respect to the LUT-based approach.

It takes benefit from those multiplexers that can invert the logic value of some CBs' inputs, such as the InvertFFinMux (see Figure 3.13). These multiplexers can be used to invert the logic value of LUTs' output thus emulating the occurrence of a *pulse* in these lines.

The steps this approach comprises, illustrated in Figure 4.13, are next presented.

a) First of all, it is necessary to check whether the LUT's output is driving some CB's input signal that can be inverted by using its associated multiplexer (trace(LUTout_j, FFin_i) $\neq \oslash$). Otherwise, this approach cannot be used.

b) The reconfiguration process just consists in changing the current state of the control bit of the suitable multiplexer to invert the logic state of the signal (MUX1_i(`'0'`/`'1'`)).

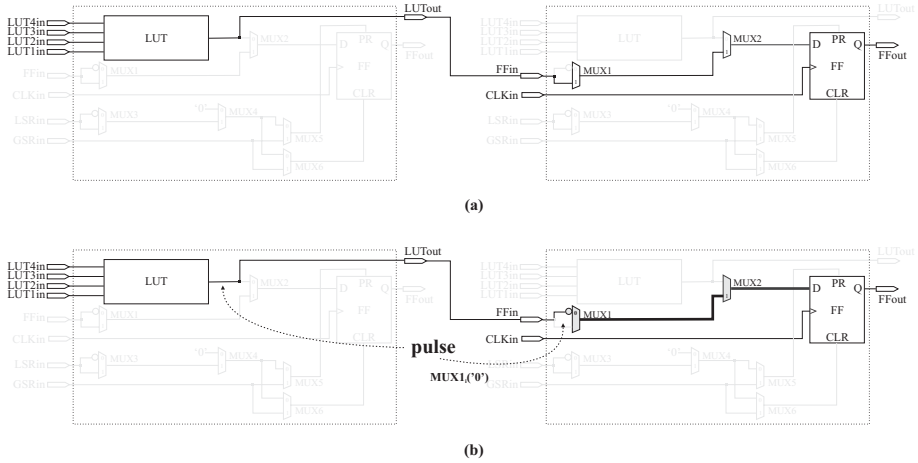

Figure 4.13: *Emulating a* pulse *into combinational logic by using inverter multiplexers. A combinational signal is routed through the FFin input of a CB (a). MUX1 is properly configured to invert the logic state of that signal, emulating the occurrence of a* pulse *in the output of the combinational circuit implemented by the LUT (b).*

When the duration of the fault expires, the fault is deleted by restoring the previous state of the multiplexer's control bit (MUX1_i(‘0’/‘1’)).

The time required to reconfigure the FPGA to inject and delete a fault when following this approach can be carried out by using Equation 4.12, where the time unit (T) represents the time spent reading/writing one bit from/to the configuration memory of the FPGA.

$$\begin{aligned} (configure\ the\ multiplexer\ to\ inject\ the\ fault) \quad & 1T+ \\ (restore\ the\ original\ configuration\ to\ delete\ the\ fault) \quad & 1T = \\ \textbf{(total)} \quad & \textbf{2T} \end{aligned} \tag{4.12}$$

4.4.3 Discussion

Equation 4.12 clearly shows that the FPGA's reconfiguration time of this novel multiplexer-based approach is constant and does not depend on any parameter of the system's model.

Figure 4.14 depicts the reconfiguration temporal cost associated to these two proposed approaches in terms of percentage of bits that have to be reconfigured (k).

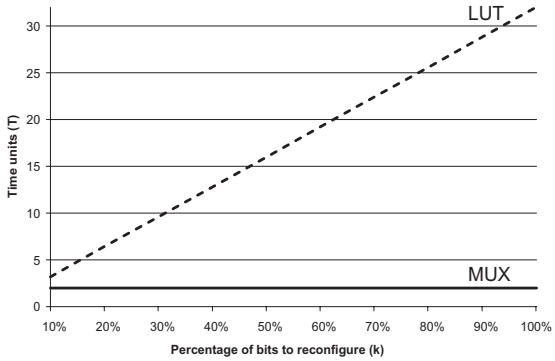

Figure 4.14: *Temporal cost of injecting a* pulse *by reconfiguring a LUT (dashed line) and by using a multiplexer to invert the combinational line (solid line).*

As can be seen in Figure 4.14 it should be advisable to use a multiplexer to inject a pulse whenever possible to increase the attainable speed-up. However, it cannot address *pulses* occurring within combinational logic implemented as a LUT.

Hence, both approaches are useful in their different contexts.

4.4.4 Summary

Pulses are transient faults that affect the combinational logic of the system.

As this logic is implemented on an FPGA by means of LUTs, the previously proposed approach for injecting *stuck-at* faults into the combinational logic of the system may be also used, with some changes, to emulate the occurrence of *pulses*.

A novel approach has also been proposed to further accelerate the insertion of these faults under some specific conditions.

Table 4.3 summarises, following a C-like pseudo-code,the process for emulating the occurrence of *pulses* by means of the different considered approaches.

4.5 Emulation of stuck-opens

Logic elements affected by *stuck-open* faults hold their previous logic value for a retention time and, after that, their state is bound to a low-logic level (‘0’).

This can be assimilated to two consecutive *stuck-at* faults. The first of them keeps the current state of the element constant until the retention time elapses. After that, a *stuck-at-0* fixes the output of the logic element to a low-logic level until the end of the experiment.

Table 4.3: *Pseudo-code for injecting and deleting a* pulse *into/from the system.*

Fault injection	**Fault deletion**

switch (target) {
  case LUT:
  stateLUT$_j$ = LUT$_j$();
  faultyLUT = LUT$_j$(location, PULSE);
  LUT$_j$(faultyLUT);
  break;
  case FFIN_LINE:
  connected = false; i = 0;
  while (!connected && (i < #FFS)) {
  segments = trace(LUTout$_j$, FFin$_i$);
  connected = (segments != null);
  if (!connected) i++;
  };
  if (connected) {
  stateMUX[1]$_i$ = MUX[1]$_i$();
  MUX[1]$_i$(not stateMUX[1]$_i$);
  }
  break;
}
	switch (target) {
	  case LUT:
	  LUT$_j$(stateLUT$_j$);
	  break;
	  case FFIN_LINE:
	  MUX[1]$_i$(stateMUX[1]$_i$);
	  break;
	}

Hence, the emulation of *stuck-opens* uses these same procedures but taking into account that the final logic value provided by the circuit depends on the current state of the affected logic element (finalValue = FF$_f$(), for instance) instead of being determined by the type of *stuck-at* fault being considered (finalValue = (fault=*stuck-at-0*) ? '0' : '1').

These faults may affect both the combinational and sequential logic of the system's model. Then, LUTs and FFs are the FPGA's configurable resources that must be considered when emulating the occurrence of this kind of faults.

This Section describes how the previously presented approaches for the emulation of *stuck-ats* may be used to inject *stuck-opens* into the circuit.

4.5.1 Injecting *stuck-opens* into combinational logic

The same procedure presented in Section 4.3.1 to fix the logic value provided by an element of a combinational circuit, is applicable in this context under the restrictions previously presented.

Figure 4.15 illustrates the use of this approach.

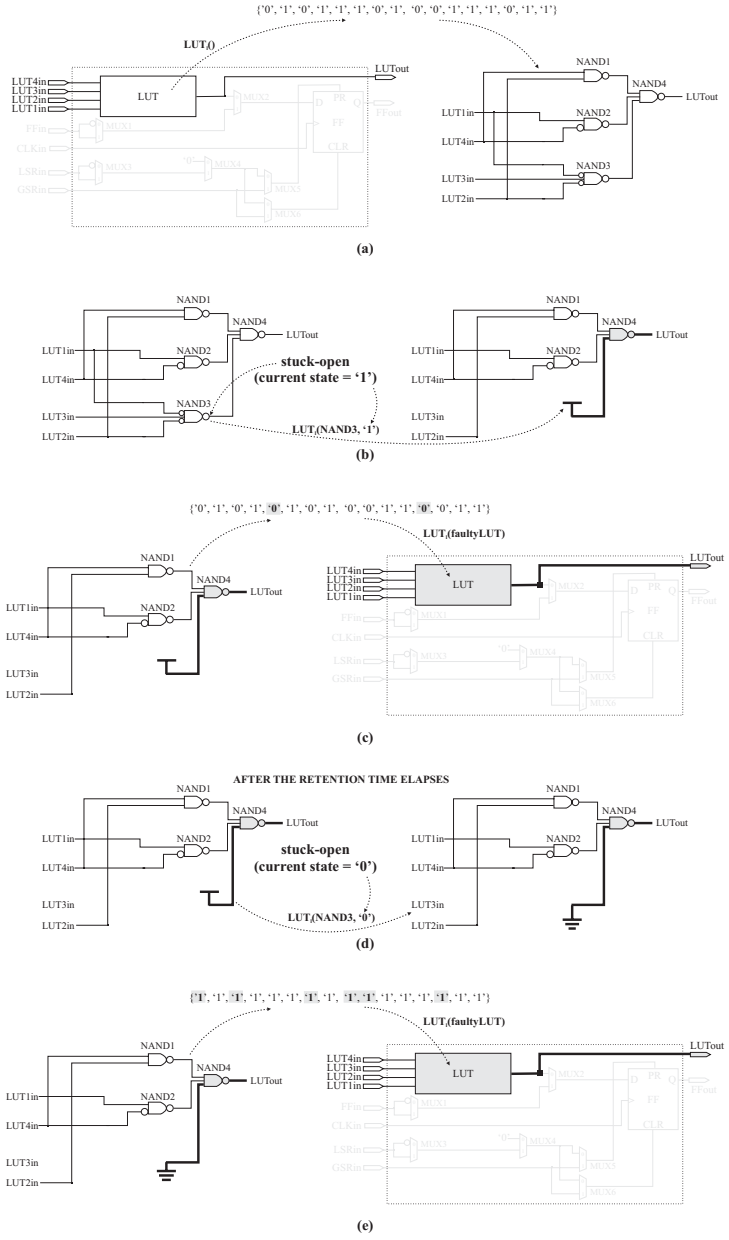

Figure 4.15: *Emulating the occurrence of a* stuck-open *into combinational logic implemented as a LUT. A structural representation of the combinational circuit is extracted from the LUT's contents (a). The output of the NAND3 gate is affected by the* stuck-open *(b) and the new truth table of the faulty circuit replaces LUT's contents (c) to keep its current state fixed. After the retention time elapses, the state of the affected element is bound to a low logic level (d) and the LUT's contents are reconfigured with the new faulty truth table (e).*

Equation 4.13 computes the time required to reconfigure the FPGA to emulate the occurrence of a *stuck-open* in terms of the percentage of bits that must be modified (k) and the time required to read and write a bit of the FPGA's configuration memory (T).

$$\begin{aligned} (change\ LUT\ to\ fix\ the\ current\ state) \quad & 16\frac{k_1}{100}T+ \\ (fix\ the\ current\ state\ to\ 0,\ if\ required) \quad & [0, 16\frac{k_2}{100}T] = \\ \textbf{(minimum total)} \quad & \mathbf{16\frac{k_1}{100}T} \\ \textbf{(maximum total)} \quad & \mathbf{16(\frac{k_1+k_2}{100})T} \end{aligned} \tag{4.13}$$

4.5.2 Injecting *stuck-opens* into sequential logic

The occurrence of *stuck-opens* into the sequential logic of the system may be emulated by following any of the three different approaches presented in Section 4.3.2. They asynchronously change the logic state of the targeted FF and hold that logic state until the end of the experiment.

This Section details the use of these approaches for emulating *stuck-opens*.

4.5.2.1 Using the unused Local Set/Reset (LSRin) line

Section 4.3.2.1 demonstrated how the logic that controls the set/reset inputs of the FF can be used to continuously activate the set/reset of the FF.

An example of the application of this procedure is shown in Figure 4.16.

The reconfiguration temporal cost of using this approach can be computed by Equation 4.14. T represents the time required to read/write one bit from/to the FPGA's configuration memory and m is the percentage of multiplexers that must be reconfigured to change and keep the state of the FF.

$$\begin{aligned} (read\ the\ FF's\ current\ state) \quad & T+ \\ (configure\ multiplexers\ to\ keep\ the\ state) \quad & (1+2\frac{m}{100})T+ \\ (reconfigure\ MUX3) \quad & T+ \\ (fix\ the\ state\ to\ 0,\ if\ required) \quad & [0, 2T] = \\ \textbf{(minimum total)} \quad & \mathbf{(3+2\frac{m}{100})T} \\ \textbf{(maximum total)} \quad & \mathbf{(5+2\frac{m}{100})T} \end{aligned} \tag{4.14}$$

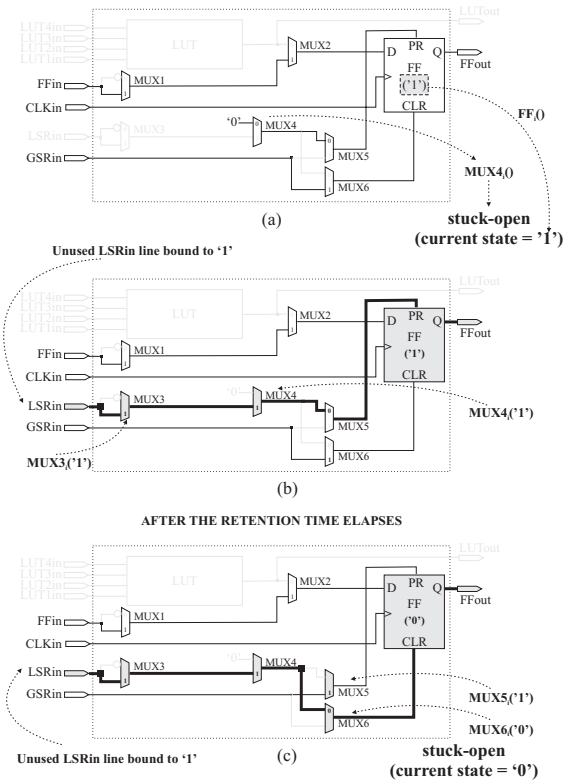

Figure 4.16: *Emulating the occurrence of a* stuck-open *into sequential logic by using the LSRin signal. The applicability of the approach is checked and the current state of the targeted FF is determined (a). The multiplexers that control the set/reset logic of the FF are reconfigured for the LSRin line to trigger the set signal of the affected FF (b) for a retention time. After that, the suitable multiplexers are reconfigured to constantly activate the reset signal of the FF (c).*

4.5.2.2 Using the unused LUT associated to the FF

The LUT that shares the CB with the targeted FF may be used to provide a constant value to the FF's input as described in Section 4.3.2.2.

This approach consists of the steps depicted in Figure 4.17.

The FPGA's reconfiguration time required to follow this approach is computed by Equation 4.15. m represents the percentage of multiplexers that must be reconfigured to invert the FF's state, whereas k indicates the percentage of bits from the LUT's contents that must be changed and T stands for the time required to read/write one bit from/to the FPGA's configuration memory.

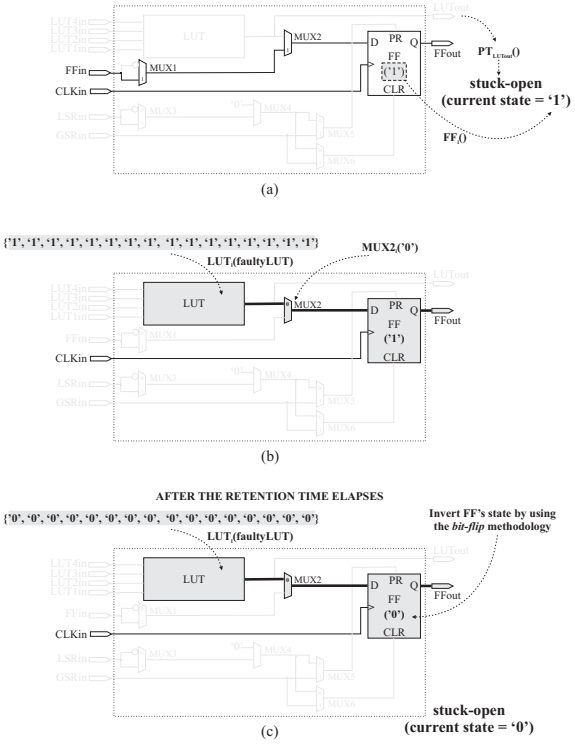

Figure 4.17: *Emulating the occurrence of a* stuck-open *into sequential logic by using the LUT associated to the affected FF. The applicability of this approach is checked (a). The contents of the LUT are reconfigured to provide the suitable constant value and its output is connected to the FF's input (b). After the retention time, The FF's state must be changed to a low logic level and the LUT is reconfigured to constantly provide that value (c).*

$$
\begin{aligned}
(read\,the\,FF's\,current\,state) \quad & T+ \\
(configure\,MUX2,\,if\,needed) \quad & [0,T]+ \\
(reconfigure\,LUT's\,contents) \quad & 16\frac{k}{100}T+ \\
(fix\,the\,state\,to\,0,\,if\,required) \quad & [\\
(invert\,the\,FF's\,state) \quad & (2+6\frac{m}{100})T+ \\
(reconfigure\,the\,LUT's\,contents) \quad & 16T \\
&] = \\
\textbf{(minimum total)} \quad & \mathbf{(1+16\frac{k}{100})T} \\
\textbf{(maximum total)} \quad & \mathbf{(20+6\frac{m}{100}+16\frac{k}{100})T}
\end{aligned}
\tag{4.15}
$$

4.5.2.3 Using the clock signal (CLKin) of the FF

Another solution to prevent the FF from changing its state, presented in Section 4.3.2.3, is to disconnect it from the clock network.

This methodology comprises the steps illustrated by Figure 4.18.

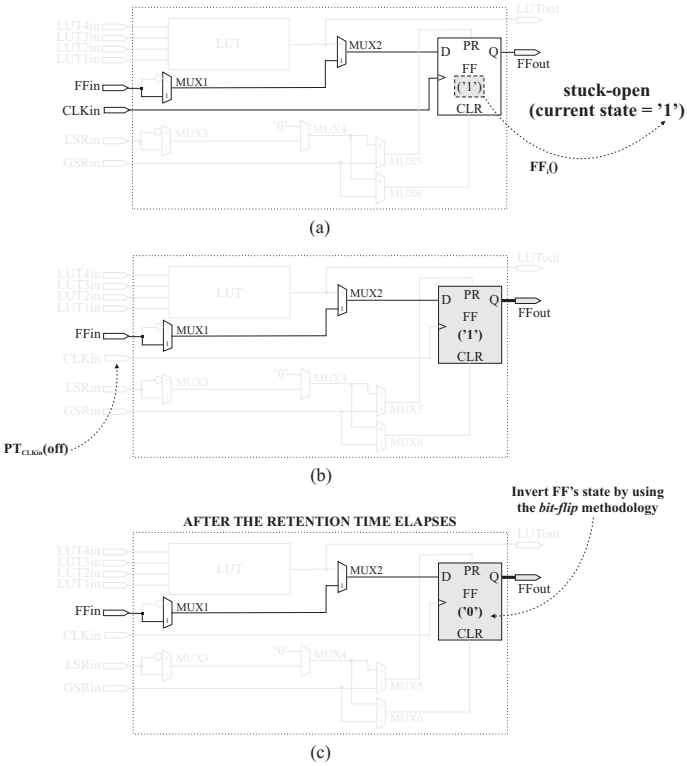

Figure 4.18: *Emulating the occurrence of a* stuck-open *into sequential logic by using the clock input signal associated to the targeted FF. The FF's current state is checked to determine the logic state it must hold (a). The FF is disconnected from the clock network by turning off the suitable pass transistor, thus preventing the FF from receiving any clock edge (b). After the retention time elapses, the state of the FF is reversed to keep a low logic level (c).*

Equation 4.16 can be use to compute the cost, in time units (T), of reconfiguring the FPGA to emulate the occurrence of a *stuck-open* following this approach. T is the time required to read/write one bit from/to the FPGA's configuration memory and m represents the percentage of multiplexers that must be reconfigured to invert the FF's current state.

$$
\begin{aligned}
(reading\ the\ FF's\ current\ state) &\quad T+ \\
(disconnecting\ the\ FF\ from\ the\ clock\ network) &\quad T+ \\
(fix\ the\ FF's\ state\ to\ 0,\ if\ needed) &\quad [0,(2+6\frac{m}{100})T]+ \\
\mathbf{(minimum\ total)} &\quad \mathbf{2T} \\
\mathbf{(maximum\ total)} &\quad \mathbf{(4+6\frac{m}{100})T}
\end{aligned}
\tag{4.16}
$$

4.5.2.4 Discussion

Three procedures, all of them based on the already proposed approaches for the emulation of *stuck-ats*, have been presented to emulate the occurrence of *stuck-opens* into sequential logic.

Figure 4.19 shows that the use of a LUT-based approach causes the longest reconfiguration times and, therefore, should be avoided whenever possible.

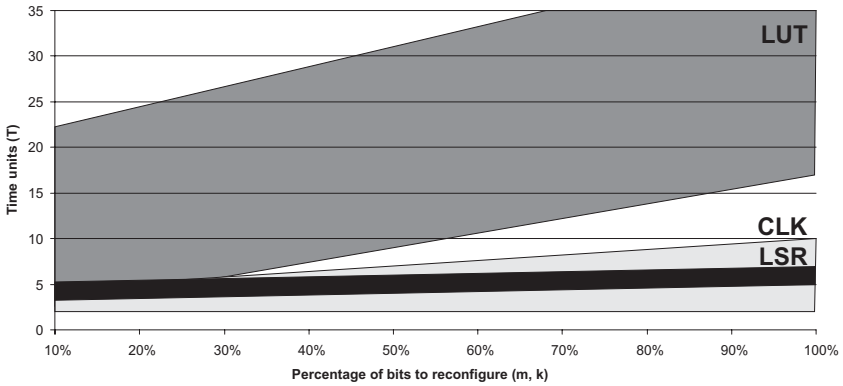

Figure 4.19: *Cost in time units of injecting a* stuck-open *into a FF by using the LSR, LUT and CLKin approaches. The cost of using a LUT, depending on the value of* m *(percentage of bits that must be reconfigured), appears in dark grey. The temporal cost of disconnecting the CLKin line is depicted in light grey. The black area indicates the temporal cost of using the LSRin signal to keep fixed the FF's faulty logic state.*

On the other hand, disconnecting the FF from the clock network presents the lowest costs in time units (T). However, when increasing the percentage of bits that must be reconfigured in order to emulate the occurrence of the fault, there exists a wide area in which the use of the LSRin-based approach may be also considered.

Hence, disconnecting the FF from the clock network is usually the best option for injecting a *stuck-open* into sequential logic.

4.5.3 Summary

Stuck-opens are permanent faults that can affect either the sequential or the combinational logic of the circuit.

They have been modelled as two consecutive *stuck-at* faults. The first one holds the current state of the affected element and, after a retention time, the second one fixes its state to '0'. Therefore, the different approaches defined for inserting *stuck-ats* into the logic of the system are also considered for the emulation of *stuck-opens*.

Table 4.4 summarises, following a C-like pseudo-code,the process for emulating the occurrence of *stuck-opens* by means of the different considered approaches.

Table 4.4: *Pseudo-code for injecting a* stuck-open *into the system.*

Fault injection

`if (target == LUT) currentState = lineState;`
`else currentState = FF`$_j$`();`
`injectStuckAt(currentState);`
`wait(retentionTime);`
`if (currentState != '0')`
`  injectStuckAt('0');`

4.6 Emulation of indeterminations

Indeterminations represent that the voltage of a logic element is between the low- and high-level thresholds for a given technology. This results in that logic element holding an undetermined logic state.

The main problem when emulating this kind of fault is that no intermediate voltage level may be induced in the configurable resources of the FPGA just by acting on its configuration memory. Although it could be possible to cause some internal short circuit or open line to obtain an indeterminate voltage value, the resulting analogue value will lead to a well defined although uncertain logic state (logic '0' or logic '1') when it goes through a buffer along the routing or at the input of a CB. Thus, we propose to emulate the occurrence of this fault by causing the affected logic element to hold the final logic state resulting from the undetermined voltage value instead of trying to generate it. As that final logic state depends on so many factors that it cannot properly be determined beforehand, we suggest to randomly generate it (rand('0', '1')).

Under these assumptions, any procedure capable of modifying the logic state of the affected elements is eligible to emulate the occurrence of *indeterminations* into the system. Thus, the very same procedures proposed in Section 4.3 to change and keep constant the logic value induced by *stuck-ats* can be also used to emulate the *indetermination* fault.

There exists three main differences when applying the procedures described in Section 4.3 in this context:

1. The final value provided by the circuit is randomly generated (finalValue = rand(‘0’, ‘1’)), instead of being determined by the type of *stuck-at* fault being considered (finalValue = (fault=*stuck-at-0*) ? ’0’ : ’1’).
2. The logic value induced by *indeterminations* may fluctuate throughout the duration of the fault. Under this assumption, the proposed emulation procedure should be followed more than once per injected fault.
3. *Indeterminations* may appear as transient or permanent faults, so the deletion of the fault must also be considered.

As this fault may affect the combinational and sequential logic of the system's model, the reconfigurable elements of the FPGA that can used to emulate the occurrence of *indeterminations* are LUTs and FFs, respectively .

Next sections summarise the previously proposed possibilities that are also useful to emulate the occurrence of *indeterminations*.

4.6.1 Injecting *indeterminations* into combinational logic

The system's combinational logic is implemented, as already explained, by means of LUTs.

Section 4.3.1 detailed the procedure to be followed to inject a *stuck-at* into combinational logic implemented via LUTs. This procedure can also be adopted to emulate the occurrence of *indeterminations*. As shown in Figure 4.20, it is necessary to change the LUT's contents for the targeted logic to generate the randomly selected final state.

This approach assumes that the faulty final state of the logic element is going to be kept constant for the duration of the fault. Otherwise, it is necessary to iterate through steps b) and c) every time the randomly generated final logic state may change due to variations in the intermediate voltage of the logic element caused, for instance, by crosstalk.

In case of emulating a transient *indetermination*, it must be deleted from the system by restoring the original contents of the LUT (LUT_l(faultFreeLUT)) after its duration elapses.

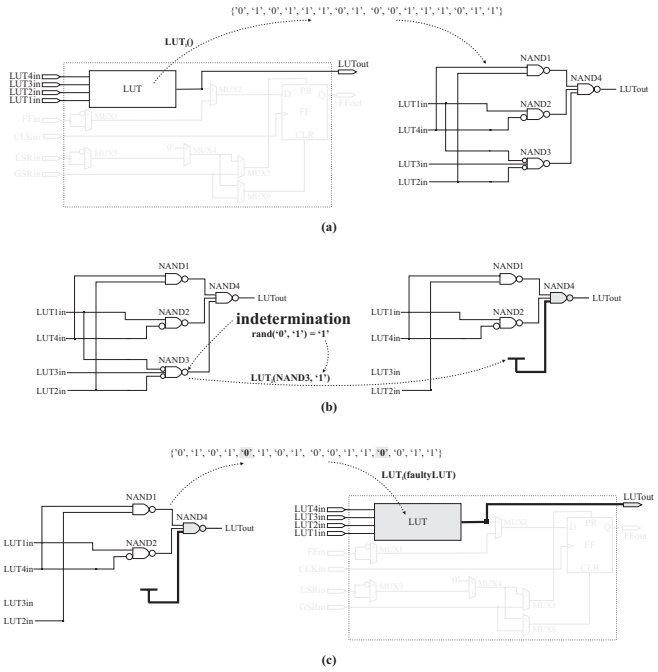

Figure 4.20: *Emulating the occurrence of an* indetermination *into combinational logic implemented as a LUT. A structural representation of the combinational circuit is extracted from the LUT's contents (a). The output of the NAND3 gate is affected by the* indetermination *(b) and the new truth table of the faulty circuit replaces LUT's contents (c) to emulate the occurrence of the fault.*

When considering that the induced logic state is constant for the duration of the fault, this approach is very similar to the one followed to inject *stuck-ats* and hence the temporal cost of this process is computed by Equation 4.7.

However, in case that the induced state changes along the duration of the fault, the FPGA's reconfiguration time depends on the number of iterations (n) throughout the emulation process and the percentage of LUT's bits that must be reconfigured in each iteration (m). Equation 4.17 computes this time expressed in time units (T, time required to read/write one bit from/to the configuration memory of the FPGA).

$$
\begin{aligned}
(change\ k\%\ of\ LUT's\ bits\ to\ inject\ the\ fault) \qquad & \sum_{i=1}^{n} 16\frac{k_i}{100}T + \\
(restore\ LUT\ contents\ to\ delete\ the\ fault) \qquad & 16\frac{k}{100}T = \qquad (4.17) \\
\textbf{(transient total)} \qquad & \mathbf{16(\frac{k}{100} + \sum_{i=1}^{n} \frac{k_i}{100})T} \\
\textbf{(permanent total)} \qquad & \mathbf{16\sum_{i=1}^{n} \frac{k_i}{100}T}
\end{aligned}
$$

4.6.2 Injecting *indeterminations* into sequential logic

The three different approaches presented in Section 4.3.2 for asynchronously changing the logic state of a FF and holding that logic state until the end of the experiment can also be used for emulating the occurrence of *indeterminations*.

These approaches are based on the use of the *LSRin* line, the use of the *LUT* associated to the targeted FF, and disconnecting the *CLKin* signal from the clock network, respectively.

4.6.2.1 Using the unused Local Set/Reset (LSRin) line

Section 4.3.2.1 demonstrated how the logic that controls the set/reset of the targeted FF may be used to continuously set/reset it, thus preventing its state from changing.

This approach is used in Figure 4.21 to inject an *indetermination* in a FF.

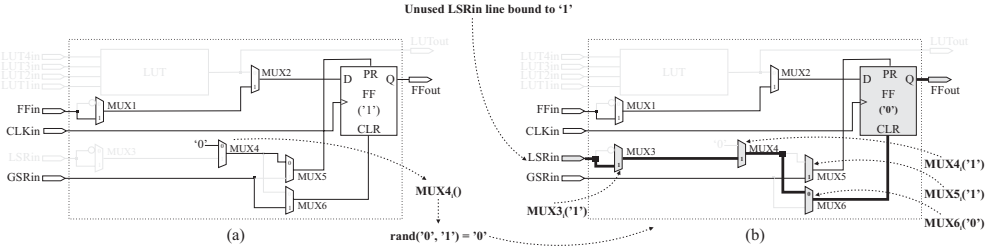

Figure 4.21: *Emulating the occurrence of an* indetermination *into sequential logic by using the LSRin signal. The configuration of the system is checked to confirm that this approach can be used (a). The multiplexers that control the set/reset logic of the FF are reconfigured for the LSRin line to feed the reset signal of the affected FF (b) thus emulating the occurrence of an* indetermination *resulting in a low-logic level.*

When considering that the logic state induced by the *indetermination* may randomly fluctuate along the duration of the fault, it is only necessary to reconfigure the multiplexers that control the set/reset logic of the FF (MUX5 and MUX6) to invert its current logic state.

When dealing with transient *indeterminations*, the fault must be deleted from the system after expiring its duration. The original configuration of the multiplexers involved in the fault injection process has to be restored (MUX3$_i$(‘0’/‘1’), MUX4$_i$(‘0’/‘1’), MUX5$_i$(‘0’/‘1’) and MUX6$_i$(‘0’/‘1’)).

Equation 4.18 computes the time required to reconfigured the FPGA to inject an *indetermination* fault under this approach, being (T) the time required to read/write one bit from/to the FPGA's configuration memory and m the percentage of multiplexers reconfigured.

$$\begin{aligned}
(configure\ the\ m\%\ of\ multiplexers) \quad & (1+2\frac{m}{100})T+ \\
(reconfigure\ MUX3) \quad & T+ \\
(restore\ m\%\ of\ multiplexers\ to\ delete\ the\ fault) \quad & (2+2\frac{m}{100})T = \\
\mathbf{(transient\ total)} \quad & \mathbf{(4+4\frac{m}{100})T} \\
\mathbf{(permanent\ total)} \quad & \mathbf{(2+2\frac{m}{100})T}
\end{aligned} \tag{4.18}$$

In case that the *indetermination* causes the logic state of the FF to change n times, the reconfiguration of the FPGA will take the time units (T) computed by Equation 4.19.

$$\begin{aligned}
(configure\ the\ m\%\ of\ multiplexers) \quad & (1+\frac{m}{100})T+ \\
(reconfigure\ MUX3) \quad & T+ \\
(reconfigure\ MUX5\ and\ MUX6\ n\ times) \quad & 2nT+ \\
(restore\ m\%\ of\ multiplexers) \quad & (2+2m)T = \\
\mathbf{(transient\ total)} \quad & \mathbf{(4+4\frac{m}{100}+2n)T} \\
\mathbf{(permanent\ total)} \quad & \mathbf{(2+2\frac{m}{100}+2n)T}
\end{aligned} \tag{4.19}$$

4.6.2.2 Using the unused LUT associated to the FF

In case that the LUT associated to the targeted FF is not currently being used, it can be reconfigured to provide a constant logic value to the FF's input as described in Section 4.3.2.2.

Figure 4.22 illustrates the use of this methodology to emulate the occurrence of an *indetermination*.

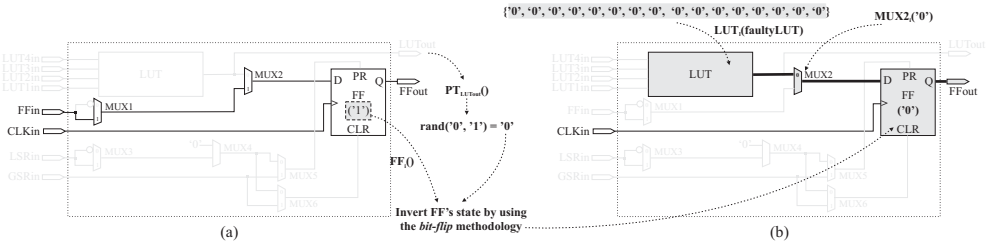

Figure 4.22: *Emulating the occurrence of an* indetermination *into sequential logic by using the LUT associated to the affected FF. The applicability of this approach is checked (a). The FF's current state must be changed by following the* bit-flip *emulation approach to match the logic state caused by the fault. The contents of the LUT are reconfigured to provide the suitable constant value and its output is connected to the FF's input (b).*

Taking into consideration that the logic state resulting from the *indetermination* may vary along the duration of the fault, it is then necessary to iterate through the steps to asynchronously change the FF's state and modify the contents of the LUT to constantly feed the FF with this new value.

When dealing with transient *indeterminations*, the FPGA has to be reconfigured again when the fault disappears to restore the previous fault-free configuration of the system.

The reconfiguration temporal cost of applying this approach is carried out by Equation 4.20. It depends on the percentage of multiplexers that must be reconfigured to invert the current state of the FF (m), on the percentage of bits to be changed from the LUT (k), and on the time required to read/write one bit from/to the FPGA's configuration memory (T).

As some of the steps might not be taken, the minimum and maximum reconfiguration times are computed.

$$
\begin{aligned}
&(read\ the\ FF's\ current\ state) && T+\\
&(invert\ the\ FF's\ state,\ if\ needed) && [0,(2+6\frac{m}{100})T]+\\
&(configure\ multiplexer\ MUX2,\ if\ needed) && [0,T]+\\
&(reconfigure\ the\ LUT's\ contents) && 16\frac{k}{100}T+\\
&(restore\ MUX2\ to\ delete\ the\ fault) && [0,T]+\\
&(restore\ LUT's\ contents\ to\ delete\ the\ fault) && 16\frac{k}{100}T=\\
&\textbf{(transient minimum total)} && \mathbf{(1+32\frac{k}{100})T}\\
&\textbf{(permanent minimum total)} && \mathbf{(1+16\frac{k}{100})T}\\
&\textbf{(transient maximum total)} && \mathbf{(5+6\frac{m}{100}+32\frac{k}{100})T}\\
&\textbf{(permanent maximum total)} && \mathbf{(4+6\frac{m}{100}+16\frac{k}{100})T}
\end{aligned}
\tag{4.20}
$$

Equation 4.21 computes the time spent reconfiguring the FPGA when the *indetermination* causes the state of the FF to change n times. The minimum and maximum values for the reconfiguration time are also provided.

$$
\begin{aligned}
&(read\ FF's\ current\ state) && T+\\
&\text{first change:} &&\\
&(invert\ state,\ if\ needed) && [0,(2+6\frac{m_0}{100})T]+\\
&(alter\ MUX2,\ if\ needed) && [0,T]+\\
&(change\ LUT's\ contents) && 16\frac{k}{100}T+\\
&\text{rest of (n-1) changes:} && \sum_{i=1}^{n-1}(\\
&(invert\ the\ FF's\ state) && (2+6\frac{m_i}{100})T+\\
&(change\ LUT's\ contents) && 16T\\
&) && +\\
&\text{delete the fault:} &&\\
&(restore\ MUX2) && [0,T]+\\
&(restore\ LUT's\ contents) && 16\frac{k}{100}T=\\
&\textbf{(transient minimum total)} && \mathbf{(32\frac{k}{100}+18n-17+\sum_{i=1}^{n-1}6\frac{m_i}{100})T}\\
&\textbf{(permanent minimum total)} && \mathbf{(16\frac{k}{100}+18n-17+\sum_{i=1}^{n-1}6\frac{m_i}{100})T}\\
&\textbf{(transient maximum total)} && \mathbf{(6\frac{m_0}{100}+32\frac{k}{100}+18n-13+\sum_{i=1}^{n-1}6\frac{m_i}{100})T}\\
&\textbf{(permanent maximum total)} && \mathbf{(6\frac{m_0}{100}+16\frac{k}{100}+18n-14+\sum_{i=1}^{n-1}6\frac{m_i}{100})T}
\end{aligned}
\tag{4.21}
$$

4.6.2.3 Using the clock signal (CLKin) of the FF

This approach, presented in Section 4.3.2.3, deals with disconnecting the FF from the clock network, thus preventing the FF to receive any clock edge.

The application of this methodology is depicted in Figure 4.23.

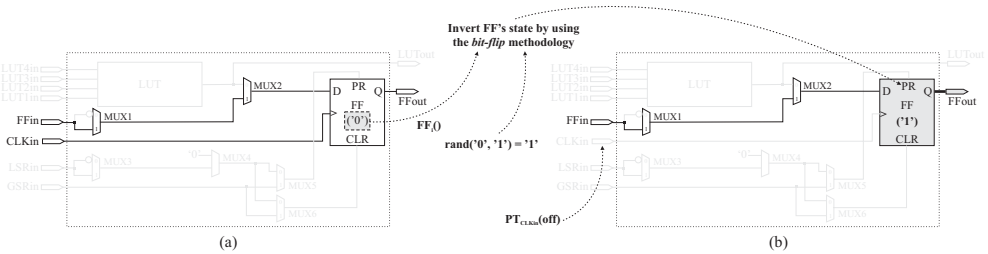

Figure 4.23: *Emulating the occurrence of an* indetermination *into sequential logic by using the clock input signal associated to the targeted FF. The FF's current state is checked to determine whether it must be reversed by following the* bit-flip *emulation approach to match the logic state caused by the fault. The FF is disconnected from the clock network by turning off the suitable pass transistor, thus preventing the FF from receiving any clock edge (b).*

If the logic state induced by the *indetermination* is supposed to fluctuate while the fault remains in the system, it is necessary to follow the procedure described in Section 4.2.2 each time that the state of the FF must be reversed.

Deleting the fault from the system, in case of dealing with transient *indeterminations*, involves reconnecting the FF to the clock network by turning on the suitable pass transistor ($\mathrm{PT}_{CLKin_t}(\mathrm{on})$).

Equation 4.22 computes the FPGA's reconfiguration time required to emulate the occurrence of an *indetermination* following this approach. m represent the percentage of multiplexers that must be reconfigured in case that the FF's state should be reversed and T stands for the time required to read/write one bit from/to the FPGA's configuration memory.

$$
\begin{aligned}
(read\,the\,FF's\,current\,state) &\quad T+ \\
(invert\,the\,FF's\,state,\,if\,needed) &\quad [0,(2+6\frac{m}{100})T]+ \\
(disconnect\,the\,FF\,\,from\,the\,clock\,network) &\quad T+ \\
(reconnect\,the\,clock\,signal\,to\,delete\,the\,fault) &\quad T= \\
\mathbf{(transient\ minimum\ total)} &\quad \mathbf{3T} \\
\mathbf{(permanent\ minimum\ total)} &\quad \mathbf{2T} \\
\mathbf{(transient\ maximum\ total)} &\quad \mathbf{(5+6\frac{m}{100})T} \\
\mathbf{(permanent\ maximum\ total)} &\quad \mathbf{(4+6\frac{m}{100})T}
\end{aligned}
\tag{4.22}
$$

The reconfiguration time for the best and worst cases when considering that the logic state induced by the *indetermination* varies n times is computed by Equation 4.23.

$$
\begin{aligned}
(read\,the\,FF's\,current\,state) &\quad T+ \\
(invert\,the\,FF's\,state,\,if\,needed) &\quad [0,(2+6\frac{m_0}{100})T]+ \\
(disconnect\,the\,clock\,network) &\quad T+ \\
(flip\,(n-1)\,times\,the\,FF's\,state) &\quad \sum_{i=1}^{n-1}(2+6\frac{m_i}{100})T+ \\
(connect\,CLK\,to\,delete\,the\,fault) &\quad T= \\
\mathbf{(transient\ minimum\ total)} &\quad \mathbf{(1+2n+\sum_{i=1}^{n-1}6\frac{m_i}{100})T} \\
\mathbf{(permanent\ minimum\ total)} &\quad \mathbf{(2n+\sum_{i=1}^{n-1}6\frac{m_i}{100})T} \\
\mathbf{(transient\ maximum\ total)} &\quad \mathbf{(3+6m_0+2n+\sum_{i=1}^{n-1}6\frac{m_i}{100})T} \\
\mathbf{(permanent\ maximum\ total)} &\quad \mathbf{(2+6m_0+2n+\sum_{i=1}^{n-1}6\frac{m_i}{100})T}
\end{aligned}
\tag{4.23}
$$

4.6.2.4 Discussion

Three different options have been proposed for emulating the occurrence of *indeterminations* into the sequential logic of a circuit.

The temporal cost associated to inject transient *indeterminations* by following these approaches is shown in Figure 4.24. Conclusions drawn from this Figure match those obtained from the study of how to emulate *stuck-ats* since they share the same procedures. However, as transient *indeterminations* must be deleted from the system, the CLKin-based approach is on average slightly better than the LSRin-based methodology.

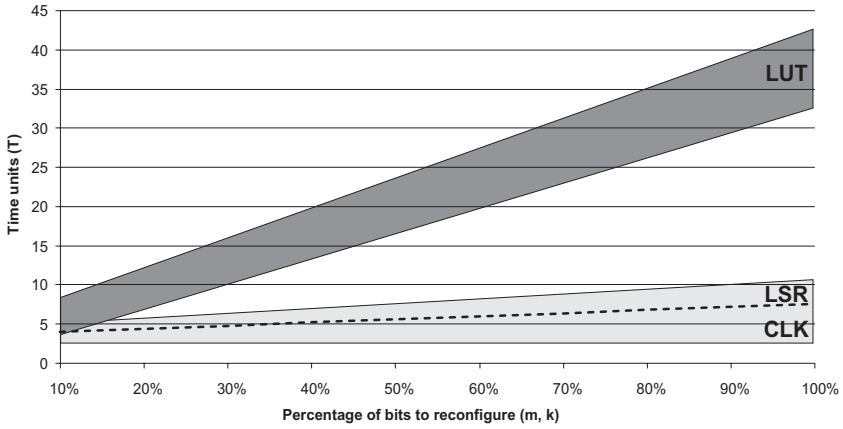

Figure 4.24: *Reconfiguration temporal cost in time units for injecting a transient* indetermination *into a FF by using the LSR, LUT and CLKin approaches. The cost of using a LUT appears in dark grey. The temporal cost of disconnecting the CLKin line is depicted in light grey. The dashed line indicates the temporal cost of using the LSRin signal to keep fixed the FF's faulty logic state.*

In case of considering that the *indetermination* can eventually modify the state of the affected FF, Figure 4.25 depicts the reconfiguration time required for each approach to change the FF's state n times. Again, the LUT-based method appears as the most time consuming method. However, the LSRin approach presents the better results in this context. The time required for the CLKin disconnection method greatly increases with n, whereas the LSRin approach increases linearly and very close to the best case for the CLKin-based method.

Hence, the best method consists in using the LSRin line, although the CLKin one is the second option when the former is not available.

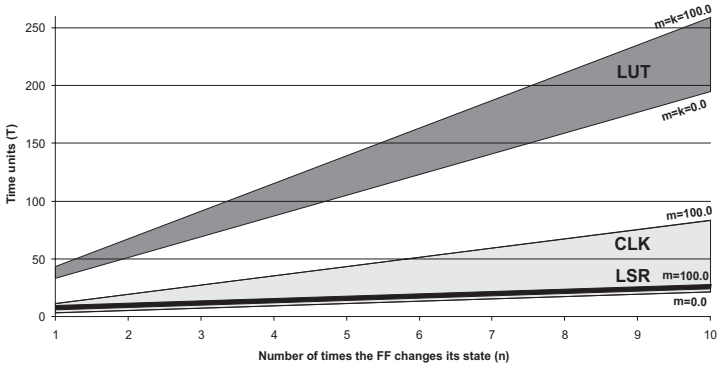

Figure 4.25: *Reconfiguration temporal cost in time units for injecting an* indetermination *into a FF by using the LSR, LUT and CLKin approaches when the state of the FF varies* n *times. The temporal cost of using a LUT is coloured in dark grey. The cost of disconnecting the CLKin line is depicted in light grey. The black area indicates the temporal cost of using the LSRin approach.*

4.6.3 Summary

Indeterminations are faults that can affect both the sequential and combinational logic of a circuit and that can have a transient or permanent duration, depending on the mechanisms that cause the occurrence of the fault.

Due to the difficulties found when trying to induce an undetermined voltage level in any element of an FPGA, the proposed solution consists in randomly determine the final logic value caused by the *indetermination*. So, all the previously defined approaches for emulating the occurrence of *stuck-ats* may also be used for injecting *indeterminations* but taking into account that: i) the final logic value induced by the fault is randomly determined, ii) that value may change along the duration of the fault due to fluctuations in the lines, and iii) in case of dealing with transient faults they must be deleted from the system after their duration expires.

Table 4.5 summarises, following a C-like pseudo-code,the process for emulating the occurrence of *indeterminations* by means of the different considered approaches.

4.7 Emulation of delays

Delays model a modification, usually an increase, in the propagation delay of a circuit which may affect its fault-free behaviour. They may appear as transient or permanent faults.

Table 4.5: *Pseudo-code for injecting and deleting an* indetermination *into/from the system.*

Fault injection	**Fault deletion**

finalValue = rand('0', '1');
switch (target) {
  case LUT:
  stateLUT$_j$ = LUT$_j$();
  faultyLUT =
  LUT$_j$(location, finalValue);
  LUT$_j$(faultyLUT);
  break;
  case FF:
  switch (approach) {
  case LSRin:
  if ((MUX[4]$_j$() == '0') ‖
  (PT$_{LSRin_j}$() == off)) {
  for (i = 3; i < 7; i++)
  stateMUX[i]$_j$ = MUX[i]$_j$();
  MUX[5]$_j$(not finalValue);
  MUX[6]$_j$(finalValue);
  MUX[4]$_j$('1');
  MUX[3]$_j$('1');
  }
  break;
  case LUT:
  if (PT$_{LUTout_j}$() == off) {
  if (FF$_j$() != finalValue)
  injectBitFlip();
  stateMUX[2]$_j$ = MUX[2]$_j$();
  stateLUT$_j$ = LUT$_j$();
  faultyLUT =
  LUT$_j$(output, finalValue);
  MUX[2]$_j$('0');
  LUT$_j$(faultyLUT);
  }
  break;
  case CLKin:
  if (FF$_j$() != finalValue)
  injectBitFlip();
  PT$_{CLKin_j}$(off);
  break;
  }
  break;
}
	switch (target) {
	  case LUT:
	  LUT$_j$(stateLUT$_j$);
	  break;
	  case FF:
	  switch (approach) {
	  case LSRin:
	  for (i = 3; i < 7; i++)
	  MUX[i]$_j$(stateMUX[i]$_j$);
	  break;
	  case LUT:
	  MUX[2]$_j$(stateMUX[2]$_j$);
	  LUT$_j$(stateLUT$_j$);
	  break;
	  case CLKin:
	  PT$_{CLKin_j}$(on);
	  break;
	  }
	  break;
	}

This fault affects the lines of the system's model and, thus, the configurable elements that must be taken into account to emulate the occurrence of a *delay* are those in charge of controlling the FPGA's routing. The Programmable Matrices (PMs) and, particularly the pass transistors (PT) that enable the interconnection among the routing segments and the CB's, are the elements that will be specially targeted.

As each logic element a line is routed through increases the delay of that line, this section presents different approaches to increase the circuit's propagation delay by adding more resources to the routing of a line.

4.7.1 Increasing the delay of a line by adding pass transistors

Most of the routing of any system implemented by means of an FPGA is done by the pass transistors that interconnect the FPGA's vertical and horizontal routing segments.

These are the most abundant configurable resources of the FPGA and, hence, it is very likely to find some unused routing segment to be added to the line affected by the fault.

Two different approaches can be distinguished when adding more routing segments (pass transistors) to the targeted line: increasing the length of the line or increasing its fan-out.

4.7.1.1 Increasing the line's length

This approach consists in adding more pass transistors along the line to increase its delay. A new segment is inserted between two consecutive segments along the current routing.

The methodology to achieve this goal, presented in Figure 4.26, can be condensed into the next steps.

a) The target line may be the output of a combinational (originPin = LUTout_l) or sequential (originPin = FFout_l) circuit.

   First of all, it is necessary to obtain the list of pass transistors the affected line is routed through (segmentsSet = trace(originPin)).

b) Two consecutive pass transistors along the routed line (source and destination) are selected as the insertion point for the new routing segment.

   These two points must be disconnected (PT_{source}(off)) to allow for the insertion of the new segment.

c) Now, it is necessary to find one or more unused segments ($\text{PT}_i() = \text{`0'}$) that can be used to connect the previously disconnected points.

These pass transistors must be reconfigured ($\text{PT}_i(\text{on})$) to establish the desired connection.

In that way, one or more pass transistors (routing segments) have been added to the line, thus increasing its length and delay. The delay introduced by these methodology and its relationship with the temporal cost associated to this process will be discussed in Sections 4.7.1.3 and 4.7.4.

It could be necessary to iterate through steps b) and c) until the required propagation delay is reached.

In case of dealing with transient *delays*, all the previous reconfigurations must be reversed when the fault disappears from the system. The new segments are unrouted ($\text{PT}_i(\text{off})$) and the original routing is restored ($\text{PT}_{source}(\text{on})$).

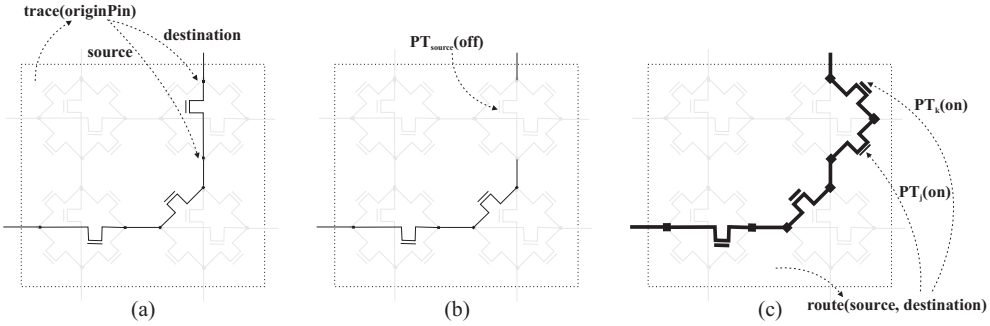

Figure 4.26: *Emulating the occurrence of a* delay *by routing the targeted line through more segments to increase its length. The two points of that line that are selected as the insertion points for the new segments (a) are unrouted (b) and the line is now routed through the new selected segments (c).*

Equation 4.24 computes the reconfiguration time, expressed in time units (T), required to apply this approach for the emulation of a *delay*. It assumes that the time required to read and write one bit from/to the FPGA's configuration memory is one time unit, whereas r represents the number of segments routed between the insertion points to increase the length of the line, and n indicates the number of iterations required to achieve the desired propagation delay.

$$
\begin{aligned}
\text{n times:} \quad & \sum_{i=1}^{n} (& & \\
(\textit{unroute the source point}) \quad & & T+ & \\
(\textit{route the } r_i \textit{ segments}) \quad & & r_i T & \\
&) & + & \\
\text{n times:} \quad & \sum_{i=1}^{n} (& & \\
(\textit{unroute the } r_i \textit{ segments to delete the fault}) \quad & & r_i T+ & \\
(\textit{reroute the source point to delete the fault}) \quad & & T & \\
&) & = & \\
\textbf{(transient total)} \quad & \mathbf{2\sum_{i=1}^{n}(1+r_i)T} & & \\
\textbf{(permanent total)} \quad & \mathbf{\sum_{i=1}^{n}(1+r_i)T} & &
\end{aligned}
\tag{4.24}
$$

4.7.1.2 Increasing the line's fan-out

The propagation delay of a line is proportional to its capacitance, which depends on the fan-out of the line. A new segment, leading nowhere, can be connected to the targeted line to increase its fan-out and consequently its propagation delay.

Considering the usually large amount of unused segments in an FPGA, this approach could applied very likely (see Figure 4.27).

a) The target line may be driven by combinational (originPin = LUTout_l) or sequential (originPin = FFout_l) circuit.

   It consists of a series of routing segments that connect its source (originPin) to another reconfigurable element (segmentsSet = trace(originPin)).

b) Once the injection point has been selected (PT_t), it is necessary to find a segment that can be connected to that point (route(PT_t, PT_s) and that is not currently driving any FPGA resource (PT_s() = '0').

   The segment is routed then by turning its pass transistor on (PT_s(on)).

   This operation increases the fan-out, and therefore the load capacitance and propagation delay, of the line without modifying its functionality. Sections 4.7.1.3 and 4.7.4 details the achieved delay in relation with the FPGA's reconfiguration time.

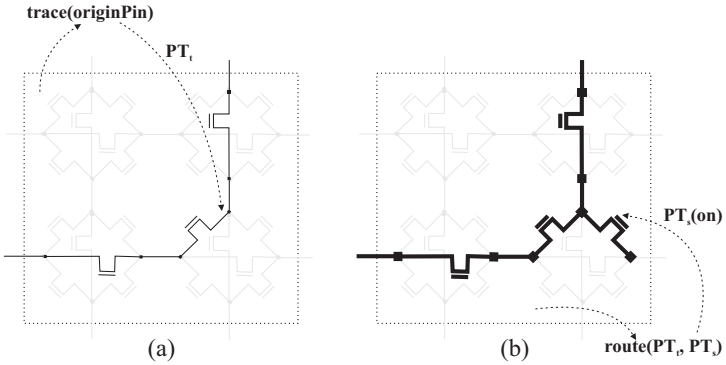

Figure 4.27: *Emulating the occurrence of a* delay *by increasing the fan-out of the line. A point of the target line (a) is connected to a new routing segment, driving no element, to increase the fan-out of the line (b).*

To further increase the propagation delay of the line, more segments can be added to augment the line's fan-out by iterating over step b).

The fault deletion process, in case of transient *delays*, consists in disconnecting all the routing segments (PT_t(off)) that have been added to restore the previous routing of the system.

The time required to reconfigure the FPGA to inject and delete a *delay* fault by using this approach is computed by Equation 4.25, where T represents the time it takes to read or write one bit of the FPGA's configuration memory.

$$\begin{aligned} (\textit{connecting the new n segments}) &\quad nT+ \\ (\textit{disconnecting the n segments to delete the fault}) &\quad nT = \\ \textbf{(transient total)} &\quad \textbf{2nT} \\ \textbf{(permanent total)} &\quad \textbf{nT} \end{aligned} \tag{4.25}$$

4.7.1.3 Discussion

Equations 4.24 and 4.25 show that increasing the fan-out of the line takes at most half of the time the other approach requires to obtain the same delay. Hence, this approach is best suited to speed-up model-based fault injection.

Taking into account that, when using a Virtex FPGA [90] for instance, a pass transistor may increase the propagation delay of a line between 0.001 and 0.018 nanoseconds, this approach is intended for injecting small propagation delays into the lines of the circuit. When dealing with large *delays* it could be advisable to use the following proposed approaches.

4.7.2 Increasing the delay of a line by adding a LUT

A usual technique used by *place and route* tools when no more segments are available is to route the line through unused LUTs. This obviously, increases the delay of the line, but helps to decongestion the routing of the system.

This same approach, depicted in Figure 4.28, can be used with the sole purpose of increasing the propagation delay of the line.

a) The *delay* may affect a line driven by combinational (originPin = LUTout_j) or sequential (originPin = FFout_j) logic.

   The LUT will be inserted between two different points from this line (source and destination) which have to be selected among the routing segments traversed by the line (segmentsSet = trace(originPin)).

   The selected LUT cannot be currently in use, otherwise another LUT should be selected. This can be guaranteed by checking that i) the LUT's output is not routed to the rest of the FPGA (PT_{LUTout_l} = off) and ii) the LUT does not drive the associated FF ($\text{MUX2}_l() \neq$ `'1'`).

b) The pass transistors of the routing segments located between the injection points (trace(source, destination)) must be turned off (PT_t(off)) to unroute them and allow for the insertion of the LUT.

c) The system has to be reconfigured to insert the chosen LUT into the line.

   The *source* point have to be routed to any of the LUT's input (input = rand(LUT1in_l, LUT2in_l, LUT3in_l, LUT4in_l)) and the LUT's output must be routed to the *destination* point. All the pass transistors along the new routing (route(source, input) and route(LUTout_l, destination)) must be turned on (PT_t(on)) to perform the connection.

   Also, the contents of the LUT have to be changed to drive the logic value of the incoming input line to its output (LUT_l(new LUT_l(input, delay))).

It is possible to further increase the delay of the line by iterating throughout that process to add more LUTs to the line's routing. Section 4.7.4 discusses the attainable delay with respect to the required reconfiguration time.

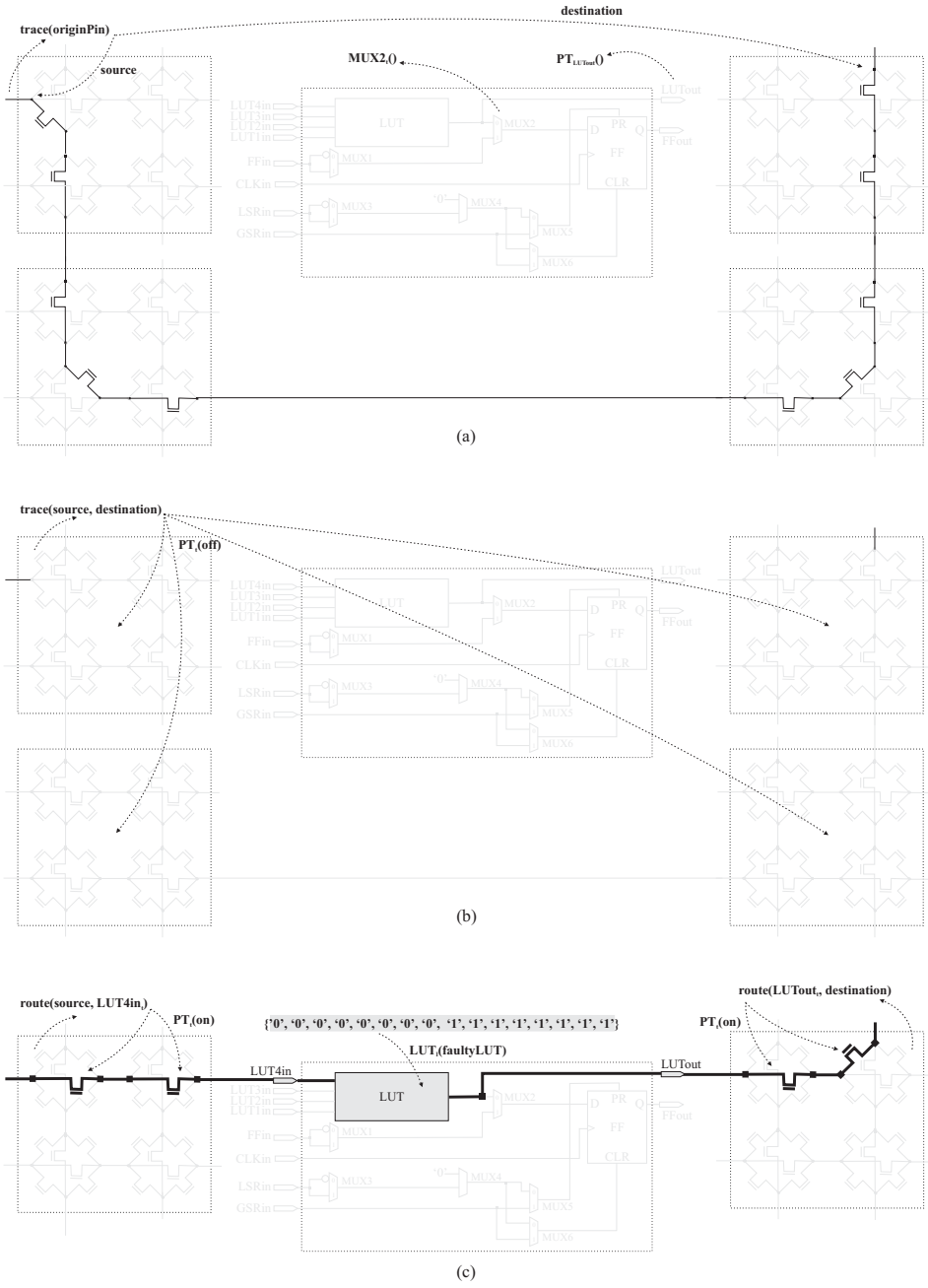

Figure 4.28: *Emulating the occurrence of a* delay *by adding a LUT to the routing. An unused LUT is selected to be inserted between two points along the routing of the line (a). These points are unrouted to allow for the LUT's insertion (b). The LUT is routed and reconfigured to drive the incoming input directly to its output (c) thus increasing the delay of the affected line.*

When dealing with transient *delays*, restoring the fault-free behaviour of the system when the fault disappears involves reconfiguring the original contents of the LUT (LUT_l(originalLUT)), unrouting the LUT (trace(source, input) and trace(LUTout_l, destination)) by turning off all the required pass transistors (PT_t(off)), and rerouting the line as it was before the fault occurrence (route(source, destination) and PT_t(on)).

Equation 4.26 computes the time units (T) required to reconfigure the FPGA to emulate the occurrence of a *delay* in the system's model. This time depends on the number of LUTs to add (n), the percentage of bits reconfigured to change the contents of the LUT (k), the number of segments routed between the insertion points (r) and the segments needed to route the LUT to these insertion points (s and t). T represents the time needed to read or write a bit to/from the configuration memory of the FPGA.

$$
\begin{aligned}
\text{n times (to inject the fault):} \quad & \sum_{i=1}^{n} (\\
(unroute\ the\ insertion\ points) \quad & r_i T + \\
(route\ source\ to\ LUT's\ input) \quad & s_i T + \\
(route\ LUT's\ output\ to\ destination) \quad & t_i T + \\
(reconfigure\ LUT's\ contents) \quad & 16\frac{k_i}{100} T \\
&) \quad + \\
\text{n times (to delete the fault):} \quad & \sum_{i=1}^{n} (\\
(reconfigure\ LUT's\ contents) \quad & 16\frac{k_i}{100} T + \\
(unroute\ LUT\ from\ destination) \quad & t_i T + \\
(unroute\ source\ from\ LUT) \quad & s_i T + \\
(reroute\ the\ insertion\ points) \quad & r_i T \\
&) \quad = \\
\textbf{(transient total)} \quad & \mathbf{2 \sum_{i=1}^{n} (r_i + s_i + t_i + 16\frac{k_i}{100}) T} \\
\textbf{(permanent total)} \quad & \mathbf{\sum_{i=1}^{n} (r_i + s_i + t_i + 16\frac{k_i}{100}) T}
\end{aligned}
\tag{4.26}
$$

This approach may increase the propagation delay of a line beyond what can be achieved by adding some pass transistors to its routing. For example, a Virtex's LUT [90] introduces between 0.29 and 0.8 nanoseconds of delay.

Hence, this is an effective methodology to moderately increase the propagation delay of a line. Some other techniques may be used to inject longer delays into the circuit.

4.7.3 Increasing the delay of a line by adding a FF

FFs are not usually used as routing resources as they modify the behaviour of the line making it synchronous.

Nevertheless, we propose applying a similar approach to that presented in the previous Section to insert a FF into the routing of the targeted line. In that way, the line will be delayed by a whole clock cycle. It may allow for the injection of longer propagations delays (n clock cycles) by inserting a n-bits shift register into the line.

This approach, illustrated by Figure 4.29, can be applied by following the steps next presented.

a) First of all, the points (source and destination) where the FF is going to be inserted, are selected among those the targeted line is routed through (segmentsSet = trace(originPin)). The line may originate from combinational (originPin = LUTout_j) or sequential (originPin = FFout_j) logic.

   Obviously, it is necessary to assure that the selected FF is not being used at all by the implemented circuit. The pass transistor that connect the FF's output to the routing of the FPGA can be checked to determine whether the it can be used to increase the delay of the line (PT_{LUTout_t}() = '0') or another FF should be found.

b) All the segments that connect the insertion points (trace(source, destination)) must be turned off (PT_t(off)) to unroute these points and enable the insertion of the FF.

c) Now, the system is reconfigured to add the FF to the line's routing.

   On one hand, all the required PTs must be turned on (PT_t(on)) to route the source point to the FF's input (route(source, FFin_f) and to route the FF's output to the destination point (route(FFout_f, destination)).

   On the other hand, the multiplexers (MUX1 and MUX2) of the CB's that holds the FF must be reconfigured for the FF to be properly inserted into the line (MUX1_f('1') and MUX2_f('1')). The FF must also be connected to the clock network (PT_{CLKin_f}(on)) to work properly.

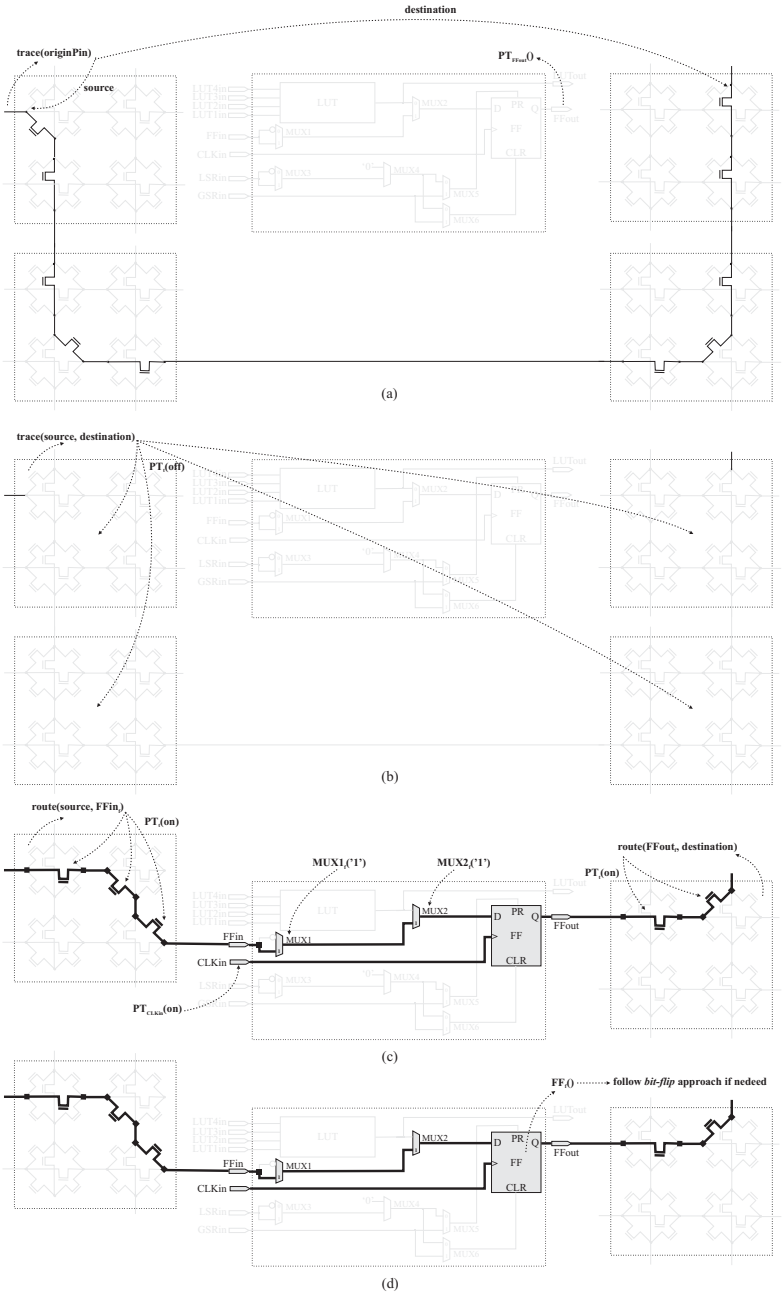

Figure 4.29: *Emulating the occurrence of a* delay *by adding a FF to the routing. The FF will be added between two points along the line's routing (a). These points are unrouted (b), and the FF is then routed to them and connected to the clock network (c). In case that the FF's state does not match that of the affected line at the injection time, the* bit-flip *approach should be followed (d).*

d) Finally, the current logic state of the FF must be obtained ($FF_f()$) to determine whether it holds the same value as the line it is now driving.

   Otherwise, the approach presented at Section 4.2.2 should be followed to reverse the current state of the FF.

This methodology could be repeatedly applied to increase the propagation delay of the line beyond one clock cycle by creating a kind of shift register.

All these changes should be undone when the transient *delay* expires.

The total reconfiguration time, expressed in time units (T), for injecting a *delay* in the system's model by following this approach can be computed by Equation 4.27. Each time unit T represents the time required to read/write a bit from/to the FPGA's configuration memory, n denotes the number of FFs to be added, r indicates the number of segments routed between the insertion points, and s and t represent the segments needed to route the FF to these insertion points. In case that the FF's state should be reversed, m represents the percentage of multiplexers that must be reconfigured.

$$
\begin{aligned}
&\text{n times (to inject the fault):} && \sum_{i=1}^{n} (\\
&(unroute\ the\ insertion\ points) && r_i T + \\
&(route\ source\ to\ FF's\ input) && s_i T + \\
&(route\ FF\ output\ to\ destination) && t_i T + \\
&(configure\ MUX1\ \&\ MUX2) && 2T + \\
&(connect\ FF\ to\ clock\ network) && T + \\
&(reverse\ FF's\ state\ if\ needed) && [0, (3 + 6\frac{m_i}{100})T] \\
& &&) \quad + \qquad (4.27) \\
&\text{n times (to delete the fault):} && \sum_{i=1}^{n} (\\
&(unroute\ clock\ network) && T + \\
&(configure\ MUX1\ \&\ MUX2) && 2T + \\
&(unroute\ FF\ from\ destination) && t_i T + \\
&(unroute\ source\ from\ FF) && s_i T + \\
&(reroute\ the\ insertion\ points) && r_i T \\
& &&) \quad = \\
&\textbf{(transient minimum total)} && \sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{6 + 2r_i + 2s_i + 2t_i})\mathbf{T} \\
&\textbf{(permanent minimum total)} && \sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{3 + r_i + s_i + t_i})\mathbf{T} \\
&\textbf{(transient maximum total)} && \sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{9 + 2r_i + 2s_i + 2t_i + 6\frac{m_i}{100}})\mathbf{T} \\
&\textbf{(permanent maximum total)} && \sum_{\mathbf{i=1}}^{\mathbf{n}} (\mathbf{6 + r_i + s_i + t_i + 6\frac{m_i}{100}})\mathbf{T}
\end{aligned}
$$

Traversing a FF in a Virtex FPGA [90], for instance, delays a signal between 0.54 and 1.4 nanoseconds. Moreover, the signal will not change until the next clock cycle, thus increasing the actual propagation delay of the line.

Hence, this technique can be exploited to inject very long propagation delays into the targeted signal.

4.7.4 Discussion

Three different approaches, mainly aimed at increasing the number of logic elements the line is routed through, have been proposed to emulate the occurrence of a *delay* in a circuit.

Table 4.6 shows the ratio between the reconfiguration time required to inject a transient *delay* and the attainable increase in the propagation delay of a line when considering a Virtex FPGA.

Table 4.6: *Comparison among the proposed approaches for emulating a* delay *in terms of the reconfiguration time (*T*) required to increase one nanosecond (*ns*) the propagation delay of the affected line when targeting a Virtex FPGA.*

Proposed approach	**Propagation delay**	**Temporal cost**	**Cost/delay ratio**
Add pass transistors:			
increasing the line's length[a]	0.03 ns	8T	266.66 T/ns
increasing the line's fan-out[b]	0.01 ns	2T	200 T/ns
Add LUTs to the line's routing[c]	0.7 ns	34T	48.57 T/ns
Add FFs to the line's routing[d]			
Minimum	1.2 ns	24T	20 T/ns
Maximum	1.2 ns	30T	25 T/ns

[a] 3 pass transistors ($m = 3$) are added to the line's routing per iteration on average.

[b] One (1) pass transistor is added in each iteration

[c] On average, 3 pass transistors ($r = s = t = 3$) are routed/unrouted, and half the contents of the LUT ($k = 8$) are reconfigured in each iteration.

[d] On average, 3 pass transistors ($r = s = t = 3$) are routed/unrouted, and half the multiplexers ($m = 0.5$) are reconfigured in each iteration.

As can be seen in Table 4.6, the insertion of FFs obtains the best cost/delay ratio. This means that this approach can inject the longest delays with the minimum reconfiguration time.

However, it is not a suitable approach for injecting small delays. It causes a minimum delay of 1.2 ns, without considering that the line is now synchronised with the clock signal, thus further delaying the signal's propagation.

LUTs can be used to inject no so long *delays* at a reasonable cost.

Although the use of pass transistors presents a very bad cost/delay ratio, it is the only effective way of introducing very small propagation delays into a line. In fact, it is a very affordable approach for injecting delays in the range of cents of nanoseconds.

Hence, all the proposed approaches have their own application field and must be taken into account when injecting *delays* into the system.

4.7.5 Summary

Delays represent a transient or permanent increase of the propagation delay of a line of circuit.

Three different approaches have been considered for the emulation of these fault. Basically, all the segments between the insertion points are unrouted to allow for the inclusion of the new logic element.

- **Pass transistors**. These elements can be added along the routing line to increase its length, or creating new branches leading nowhere to increase its fan-out. They can inject very small delays.
- **Look-Up Tables**. LUTs are also used as routing elements when no more possibilities are available for the *place and route* tool. They can be used to moderately increase the propagation delay of the line.
- **Flip-Flops**. They may cause the longest delays, reaching up to a clock cycle for each inserted FF following a shift register pattern.

Table 4.7 summarises, following a C-like pseudo-code,the process for emulating the occurrence of *delays* by means of the different considered approaches.

4.8 Emulation of shorts

The *short* models a modification in the routing of the system that results in the interconnection of two different lines of the circuit.

As this fault affects the routing of the system's model, the configurable elements that must be considered for emulating the occurrence of *shorts* are those that control the routing of the FPGA. Pass transistors (PT), which enable the interconnection among the routing segments and the CBs, are the elements that may be reconfigured to inject and delete this kind of fault.

Table 4.7: *Pseudo-code for injecting and deleting a* delay *into/from the system.*

Fault injection

```
if (origin == LUT) originPin = LUTout_j;
else originPin = FFout_j;
segmentsSet = trace(originPin);
source = rand(segmentsSet);
destination = rand(segmentsSet);
switch (target) {
   case PT:
     neighSet = neighbours(source);
     found = false; i = 0;
     while (!found && (i < neighSet.length)) {
       destination = (PT) neighSet[i];
       if (approach == LENGTH)
         found = ((destination() == on) &&
           (destination in segmentsSet));
       else
         found = ((destination() == off) &&
           !(destination in segmentsSet));
       if (!found) i++;
     }
     if (found) {
       switch (approach) {
         case LENGTH:
           newRouting = route(source, destination);
           if (newRouting != null) {
             destination(off);
             for (i = 0; i < newRouting.length; i++)
               ((PT) newRouting[i])(on);
           }
           break;
         case FANOUT:
           destination(on);
           break;
       }
     }
     break;
   case LUT:
   case FF:
     found = false; i = 0;
     size = (target == LUT) ? #LUTS : #FFs;
     while (!found && (i < size)) {
       if (target == LUT) {
         if ((PT_{LUTout_i}() == off) && (MUX[2]_i() == '0')) {
           inPin = rand(LUT1in_i, LUT2in_i, LUT3in_i, LUT4in_i);
           srcRouting = route(source, inPin);
           dstRouting = route(LUTout_i, destination);
         }
       } else {
         if (PT_{FFout_i}() == off) {
           srcRouting = route(source, FFin_i);
           dstRouting = route(FFout_i, destination);
         }
       }
       if ((srcRouting != null) && (dstRouting != null))
         found = true;
       else i++;
     }
     if (found) {
       oldRouting = trace(source, destination);
       for (k = 0; k < oldRouting.length; k++)
         ((PT) oldRouting[k])(off);
       for (k = 0; k < srcRouting.length; k++)
         ((PT) srcRouting[k])(on);
       for (k = 0; k < dstRouting.length; k++)
         ((PT) dstRouting[k])(off);
       if (target == LUT) {
         stateLUT_i = LUT_i();
         faultyLUT = LUT_i(inPin, delay);
         LUT_i(faultyLUT);
       } else {
         stateMUX[1]_i = MUX[1]_i(); stateMUX[2]_i = MUX[2]_i();
         MUX[1]_i('1'); MUX[2]_i('1'); PT_{CLKin_i}(on);
         if (FF_i() != lineState) injectBitFlip();
       }
     }
     break;
}
```

Fault deletion

```
switch (target) {
   case PT:
     switch (approach) {
       case LENGTH:
         for (i = 0; i < newRouting.length; i++)
           ((PT) newRouting[i])(off);
         destination(on);
         break;
       case FANOUT:
         destination(off);
         break;
     }
     break;
   case LUT:
   case FF:
     for (k = 0; k < srcRouting.length; k++)
       ((PT) srcRouting[k])(off);
     for (k = 0; k < dstRouting.length; k++)
       ((PT) dstRouting[k])(off);
     for (k = 0; k < oldRouting.length; k++)
       ((PT) oldRouting[k])(on);
     if (target == LUT)
       LUT_i(stateLUT_i);
     else {
       MUX[1]_i(stateMUX[1]_i);
       MUX[2]_i(stateMUX[2]_i);
       PT_{CLKin_i}(off);
     }
     break;
}
```

The occurrence of a *short* causes an increase in the current flow that may damage the affected circuit.

As reported by [91] after extensive experiments, FPGAs will not suffer any damage as a result of a short-circuit between two of their routed lines. Fortunately, the impedance of the routing switches used to interconnect the targeted lines limits the increase of the current due to the short-circuit.

However, the occurrence of multiple simultaneous *shorts* may increase the current beyond the limit supported by the device, resulting in its possible destruction. The number of simultaneous faults that will eventually damage the programmable device was determined by [92] and set at about fifty.

Therefore, there exists no real risk when short-circuiting only two lines of an FPGA to emulate the occurrence of one *short* into the system.

This can be achieved by following the steps depicted in Figure 4.30.

a) The targeted lines may be driven by combinational (originPin = LUTout_l) or sequential elements (originPin = FFout_f).

   First of all, the list of pass transistors that each line is routed through must be obtained (segmentsSet = trace(originPin)).

b) Two points from that lists are selected as the segments (source and destination) that will be connected to inject the fault.

   In case that a possible routing between these two points exists it is necessary to turn on (PT_t(on)) all the pass transistors along that path (route(source, destination)) to perform the connection.

   Now, the system behaves as if a *short* has affected these two lines. Hence the emulation of *shorts* is basically reduced to a routing problem.

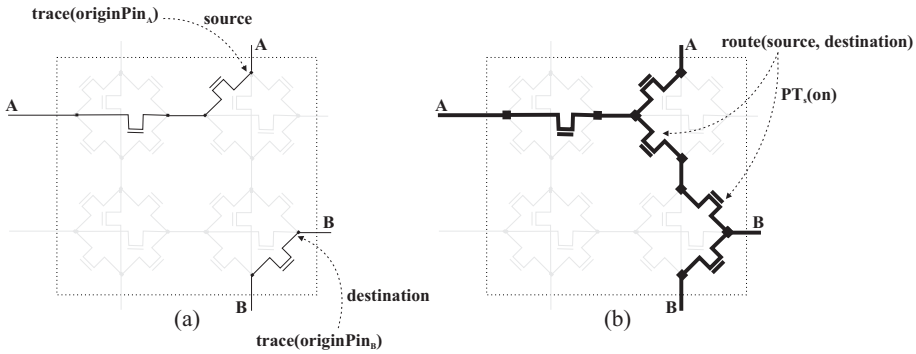

Figure 4.30: *Emulating the occurrence of a* short *between two lines of the circuit. A point from each targeted line is selected as an interconnection point (a). These two points are routed thus short-circuiting the lines together (b).*

The time devoted to reconfigure the FPGA to emulate a *short*, computed by Equation 4.28, depends on the number of pass transistors (m) needed to interconnect the targeted lines and the time (T) required to read/write one bit from/to the FPGA's configuration memory.

$$\begin{array}{ll} (routing\ m\ segments) & mT = \\ \textbf{(total)} & \textbf{mT} \end{array} \tag{4.28}$$

Table 4.8 summarises, following a C-like pseudo-code,the process for emulating the occurrence of *shorts* by means of the proposed approach.

Table 4.8: *Pseudo-code for injecting a* short *into the system.*

Fault injection

`if (origin`$_j$ `== LUT) originPin`$_j$ `= LUTout`$_j$`;`
`else originPin`$_j$ `= FFout`$_j$`;`
`if (origin`$_k$ `== LUT) originPin`$_k$ `= LUTout`$_k$`;`
`else originPin`$_k$ `= FFout`$_k$`;`
`segmentsSet`$_j$ `= trace(originPin`$_j$`);`
`segmentsSet`$_k$ `= trace(originPin`$_k$`);`
`source = rand(segmentsSet`$_j$`);`
`destination = rand(segmentsSet`$_k$`);`
`newRouting = route(source, destination);`
`if (newRouting != null) {`
`  for (i = 0; i < newRouting.length; i++)`
`  ((PT) newRouting[i])(on);`
`}`

4.9 Emulation of open-lines

Open-lines result from problems in the routing of the circuit that break up a line interconnecting different logic elements into two separated segments.

This fault deals with the routing of the system's model and, therefore, the reconfigurable elements of the FPGA that can be used for emulating the occurrence of an *open-line* are those that perform the interconnection among the different logic elements. Hence, any pass transistor (PT) along the routing of the selected line can be used to split it up into two isolated segments.

The next points describe the proposed procedure to emulate the occurrence of an *open-line* into the system.

a) The targeted line may be driven by combinational (originPin = LUTout_l) or sequential logic elements (originPin = FFout_f). The first step consists in obtaining the set of pass transistors the targeted line passes through (trace(originPin)).

   One of these segments is selected as the injection point for the *open-line*.

b) The control bit of the targeted pass transistor must be reconfigured to turn the transistor off (PT_t(off)).

   In that way, the line is divided into two segments, thus emulating the occurrence of an *open-line*.

   It is to note that the effects of that fault on the system will greatly depend on the targeted pass transistor, since the logic elements that will not be driven anymore may vary according to the selected pass transistor.

This approach is depicted in Figure 4.31 where three possible injection points are located, resulting in different *open-lines*.

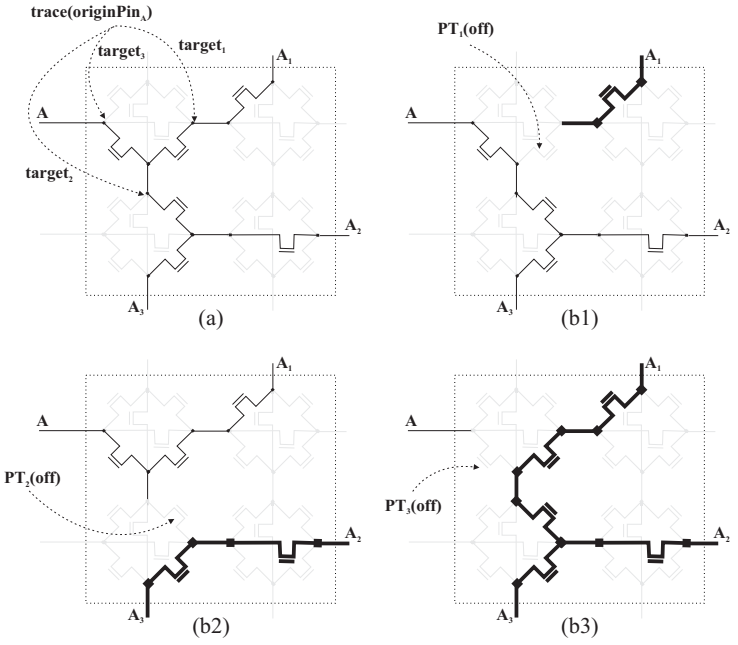

Figure 4.31: *Emulating the occurrence of an* open-line *in a line of the circuit. One segment of the line is selected as the point where the line will break up. Three different points of interest have been considered in this example (a). The first of the selected pass transistors is turned off resulting in an* open-line *affecting the line segment* A_1 *(b1). In this case, both the segments* A_2 *and* A_3 *are affected by the disconnection of the second pass transistor (b2). Finally, the three segments the line A drives can be affected by the fault by turning off the third of the considered pass transistors (b3).*

Equation 4.29 determines the time needed to reconfigure the FPGA to emulate the occurrence of *open-lines*, which only depends on the time (T) required to read/write one bit from the FPGA's configuration memory.

$$\begin{array}{rll} (\textit{unrouting the targeted segment}) & T = & \\ \textbf{(total)} & \mathbf{T} & \end{array} \tag{4.29}$$

Table 4.9 summarises, following a C-like pseudo-code,the process for emulating the occurrence of *open-lines* by means of the proposed approach.

Table 4.9: *Pseudo-code for injecting an* open-line *into the system.*

Fault injection

`if (origin == LUT) originPin = LUTout`$_j$`;`
`else originPin = FFout`$_j$`;`
`segmentsSet = trace(originPin);`
`target = rand(segmentsSet);`
`target(off);`

4.10 Emulation of bridgings

Bridging models the occurrence of a special combination of *short* and *open-line*: one of the lines is affected by an *open-line* and one of the resulting segments is connected to the other line (*short*).

As a combination of *shorts* and *open-lines*, the *bridging* is a fault model affecting the routing of the system's model. Hence, the reconfigurable elements of the FPGA that must be taken into account for its emulation are the pass transistors (PT) that are in charge of interconnecting the routing segments.

Emulating the occurrence of a *bridging*, as shown in Figure 4.32, comprises the execution of the following steps.

a) The lines that are going to be affected by the fault may be driven by combinational (originPin = LUTout$_l$) or sequential logic (originPin = FFout$_f$) indistinctly. The first step consists in determining the set of pass transistors each of the targeted lines is routed through (trace(originPin)).

   This is all the information required to decide at which point of one of the lines an *open-line* occurs (target$_{open}$), and which are the points (source and destination) that will be connected by the *short*.

b) As previously explained in Section 4.9, the occurrence of an *open-line* may be emulated by turning off one of the pass transistors the line is routed through.

Thus, modifying the configuration of the control bit of the targeted transistor ($\mathrm{PT}_{target_{open}}(\mathrm{off})$) will result in the line being divided into two different segments.

c) Now, the source and destination points must be routed to emulate the occurrence of a *short* as presented in Section 4.8.

Once a possible route between these two points is found (route(source, destination)), all the pass transistors along that path must be turned on ($\mathrm{PT}_t(\mathrm{on})$) to perform the connection between these points.

The effect of this *bridging* on the system will depend on the particular points and lines selected for injecting the *open-line* and the *short*.

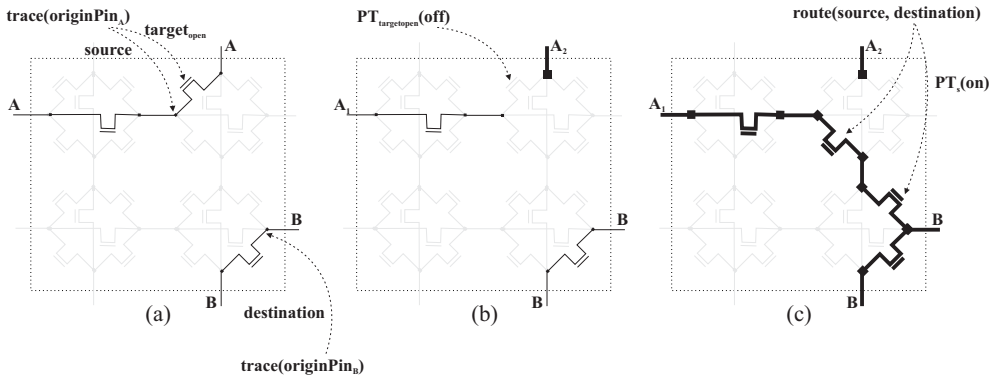

Figure 4.32: *Emulating the occurrence of a* bridging *between two lines of the circuit. One point from each targeted line is selected as an interconnection point, and another segment is selected as the point where line A will break up (a). The selected pass transistors is turned off resulting in an* open-line *affecting the line A (b). The interconnection points are routed thus short-circuiting line B and the segment A_1 (c).*

The time this reconfiguration takes, which depends on the number (m) of pass transistors that must be reconfigured to interconnect the targeted lines and the time (T) required to read/write on bit from/to the FPGA's configuration memory, can be computed by Equation 4.30.

$$\begin{array}{rrl} (unrouting\ the\ targeted\ segment) & T & \\ (routing\ m\ segments\ to\ connect\ both\ lines) & mT = & (4.30) \\ \textbf{(total)} & \mathbf{(1+m)T} & \end{array}$$

Table 4.10 summarises, following a C-like pseudo-code,the process for emulating the occurrence of *bridgings* by means of the proposed approach.

Table 4.10: *Pseudo-code for injecting a* bridging *into the system.*

Fault injection

```
if (origin_j == LUT) originPin_j = LUTout_j;
else originPin_j = FFout_j;
if (origin_k == LUT) originPin_k = LUTout_k;
else originPin_k = FFout_k;
segmentsSet_j = trace(originPin_j);
segmentsSet_k = trace(originPin_k);
target = rand(segmentsSet_j);
neighSet = neighbours(target);
sourceSet = emptySet(); i = 0;
while (i < neighSet.length) {
   source = (PT) neighSet[i];
   if ((source() == on) &&
     (source in segmentsSet_j));
       addElement(source, sourceSet);
   i++;
}
if (sourceSet != null) {
   source = rand(sourceSet);
   destination = rand(segmentsSet_k);
   newRouting = route(source, destination);
   if (newRouting != null) {
     target(off);
     for (i = 0; i < newRouting.length; i++)
       ((PT) newRouting[i])(on);
   }
}
```

4.11 Conclusions

Advances in semiconductor technologies are not only increasing the likelihood of occurrence of transient faults, but are also extending the spectrum of transient and permanent fault models that have to be considered when assessing the dependability of deep-submicron manufactured systems beyond the classical *bit-flip* and *stuck-at*.

- **Bit-flip**. It models an inversion in the logic state held by a memory cell. This faulty state remains there until the cell is rewritten.
- **Stuck-at**. A logic element keeps constant the current value of its output regardless its inputs.
- **Pulse**. A combinational element targeted by this fault momentarily inverts its current logic state for the duration of the fault modelling a Single Event Transient (SET).
- **Stuck-open**. The affected element holds its current logic level for a retention time, which is bound to `'0'` afterwards.
- **Indetermination**. It represents an undetermined voltage value between the high- and low-level threshold for a given technology. It results in an unknown logic state.
- **Delay**. It models an increase in the propagation delay of a line which may affect the behaviour of the system.
- **Short**. It models a short-circuit between two of the lines of the circuit.
- **Open-line**. It assumes that a line of the circuit breaks up, being divided into two different segments.
- **Bridging**. This is a special combination of *short* and *open-line*. Four different cases have been identified.

The main difference between the proposed methodologies for the injection of permanent and transient faults is that permanent faults remain into the system. Although transient faults disappear from the system after a short period of time their effects may remain there longer.

As following a Run-Time Reconfiguration approach, the injection of faults into the system is closely related to the architecture of the underlying FPGA. Therefore, the fault injection process for each fault model have been described using the generic FPGA architecture presented in Section 3.4.

Different approaches have been proposed for injecting each considered fault into the system's model. All of them have been described in terms of which elements of the system's model can be targeted by the fault, which reconfigurable elements of the FPGA they map to, and how the configuration memory of the FPGA should be modified to reconfigure all these elements to emulate the behaviour of the system in the presence of that particular fault.

Although this methodology is generic and flexible enough to emulate a wide range of different possibilities, only those faults previously described have been considered for emulation. For instance, transient versions of the studied permanent faults could be easily emulated by following the proposed approaches.

The time required to reconfigure the FPGA to inject the fault, and delete it after its duration elapses in case of transient faults, has also been calculated to enable the comparison among the different proposed approaches.

Table 4.11 summarises all the different approaches that have been proposed for emulating the occurrence of these faults.

This study shows how FPGAs can be used to emulate a wide range of fault models considered representative of new deep-submicron technologies. Moreover, the different proposed approaches are expected to accelerate model-based transient fault injection experiments according to their strengths and weaknesses.

It must be noted that these generic proposals should be adapted afterwards to the particular architecture of each FPGA family, as Section 5.4.4 shows for the Virtex family. More insights regarding the attainable speed-up are provided in Chapter 5.

Table 4.11: *Proposed approaches for fault emulation.*

Fault model	Target	Description	Cost function[a]
Bit-flip	Memory cells	Invert the BRAM's bit	$2T$
	Registers	Invert the FF's state by using the GSRin line	$(n + 4n\frac{m}{100})T$
		Invert the FF's state by using the LSRin line	$(3 + 6\frac{m}{100})T$
Stuck-at	Comb. logic	Recompute the truth table of the LUT-implemented circuit	$16\frac{k}{100}T$
	Registers	Fix the FF's state by using the LSRin line	$(2 + 2\frac{m}{100})T$
		Fix the FF's state by using the free LUT	$[(1 + 16\frac{k}{100})T, (4 + 6\frac{m}{100} + 16\frac{k}{100})T]$
		Fix the FF's state by using the CLKin line	$[2T, (4 + 6\frac{m}{100})T]$
Pulse	Comb. logic	*stuck-at* LUT-based approach	$32\frac{k}{100}T$
		Invert LUT's output by using a multiplexer at a CB's input	$2T$
Stuck-open	Comb. logic	*stuck-at* LUT-based approach	$[16\frac{k_1}{100}T, 16(\frac{k_1+k_2}{100})T]$
	Registers	*stuck-at* LSRin-based approach	$[(3 + 2\frac{m}{100})T, (5 + 2\frac{m}{100})T]$
		stuck-at LUT-based approach	$[(1 + 16\frac{k}{100})T, (20 + 6\frac{m}{100} + 16\frac{k}{100})T]$
		stuck-at CLKin-based approach	$[2T, (4 + 6\frac{m}{100})T]$
Indetermination	Comb. logic	*stuck-at* LUT-based approach	$TR = 16(\frac{k}{100} + \sum_{i=1}^{n}\frac{k_i}{100})T$
			$P = 16\sum_{i=1}^{n}\frac{k_i}{100}T$
	Registers	*stuck-at* LSRin-based approach	$TR = (4 + 4\frac{m}{100} + 2n)T$
			$P = (2 + 2\frac{m}{100} + 2n)T$
		stuck-at LUT-based approach	$TR = [(32\frac{k}{100} + 18n - 17 + \sum_{i=1}^{n-1} 6\frac{m_i}{100})T,$ $(6\frac{m_0}{100} + 16\frac{k}{100} + 18n - 14 + \sum_{i=1}^{n-1} 6\frac{m_i}{100})T]$
			$P = [(16\frac{k}{100} + 18n - 17 + \sum_{i=1}^{n-1} 6\frac{m_i}{100})T,$ $(6\frac{m_0}{100} + 16\frac{k}{100} + 18n - 14 + \sum_{i=1}^{n-1} 6\frac{m_i}{100})T]$
		stuck-at CLKin-based approach	$TR = [(1 + 2n + \sum_{i=1}^{n-1} 6\frac{m_i}{100})T,$ $3 + 6m_0 + 2n + \sum_{i=1}^{n-1} 6\frac{m_i}{100})T]$
			$P = [(2n + \sum_{i=1}^{n-1} 6\frac{m_i}{100})T,$ $(2 + 6m_0 + 2n + \sum_{i=1}^{n-1} 6\frac{m_i}{100})T]$
Delay	Routing logic	Increase PTs along the line	$TR = 2\sum_{i=1}^{n}(1 + r_i)T$
		Cost/Delay ratio = 266.66 T/ns	$P = \sum_{i=1}^{n}(1 + r_i)T$
		Increase line's fan-out Cost/Delay ratio = 200 T/ns	$TR = 2nT$ $P = nT$
		Reroute to pass through LUTs	$TR = 2\sum_{i=1}^{n}(r_i + s_i + t_i + 16\frac{k_i}{100})T,$
		Cost/Delay ratio = 48.57 T/ns	$P = \sum_{i=1}^{n}(r_i + s_i + t_i + 16\frac{k_i}{100})T$
		Reroute to pass through FFs Cost/Delay ratio = [20, 25] T/ns	$TR = [\sum_{i=1}^{n}(6 + 2r_i + 2s_i + 2t_i)T,$ $\sum_{i=1}^{n}(9 + 2r_i + 2s_i + 2t_i + 6\frac{m_i}{100})T]$
			$P = [\sum_{i=1}^{n}(3 + r_i + s_i + t_i)T,$ $\sum_{i=1}^{n}(6 + r_i + s_i + t_i + 6\frac{m_i}{100})T]$
Short	Routing logic	Connect lines using PTs	mT
Open-line	Routing logic	Break line turning off PTs	T
Bridging	Routing logic	*short* and *open-line*	$(1 + m)T$

[a] TR = transient fault, P = permanent fault

Chapter 5

FADES: a Tool Supporting FPGA-Based Fault Injection

*In the previous Chapter, different approaches have been proposed for emulating the occurrence of transient (*bit-flip*,* pulse*,* indetermination*, and* delay*) and permanent (*stuck-at*,* stuck-open*,* indetermination*,* delay*,* short*,* open-line*, and* bridging*) hardware faults considered representative of deep-submicron manufactured systems. These approaches have been defined in terms of a generic FPGA architecture and, thus, should be adapted to the particular architecture of each family of FPGAs.*

This Chapter presents the first prototype of an FPGA-based fault injection tool that follows a Run-Time Reconfiguration approach to emulate the behaviour of the system in the presence of the studied faults. The JBits package, provided by Xilinx, is used to fit the whole set of considered fault models into the Xilinx's Virtex architecture. The architecture of this tool is described along with the different processes required to conduct a fault injection campaign.

5.1 Introduction

FPGA-based fault injection tools consist of two basic elements identified as i) the FPGA in charge of implementing the model of the system under study, and ii) the software tool that manages the execution of fault injection experiments.

Run-Time reconfiguration techniques are closely related to the architecture of the FPGAs family selected to implement the model of the system. Having this in mind, Chapter 4 has described how to emulate the whole set of fault models considered representative of new deep-submicron technologies using the generic FPGA architecture presented in Section 3.4.

The different proposed approaches for emulating each one of the considered fault models must be adapted for its implementation onto the particular architecture of the selected FPGA. Moreover, it is necessary to take into account the support and tools the device manufacturer provides for managing the reconfiguration memory of that particular family of FPGAs.

Therefore, choosing a particular FPGAs family could be very important for properly implementing all the proposed approaches.

The application that manages the experimentation should implement the proposed methodology for injecting the whole set of considered hardware fault models. Three different process have been identified for the emulation of both transient and permanent faults: i) the *definition* process consists in obtaining all the required parameters to specify the fault injection experiments to be performed; ii) the *execution* process must perform all the requested fault injection experiments while monitoring the system's execution; and iii) the *analysis* process tries to determine the impact of these faults into the system's behaviour.

The selection of the FPGAs family will also influence the way this application should obtain all the information required to emulate the faults, modify the configuration memory of the FPGA to inject and delete them, and collect data to draw conclusions from the available results.

This Chapter presents a fault emulation tool named FADES (***F****PGA-based Framework for the* ***A****ssessment of the* ***D****ependability of* ***E****mbedded* ***S****ystems*). The main goals this tool tries to achieve are i) coping with the three key processes that have been identified for the management of fault emulation experiments, ii) implementing the different possibilities previously described to emulate all the considered fault models onto a particular family of FPGAs following a Run-Time Reconfiguration approach, and iii) keeping this tool as simple as possible for non-skilled users to be able to use it.

The architecture of FADES is described in Section 5.2 along with the selection process that lead us to choose the Virtex FPGA family from Xilinx as the most suitable architecture for implementing the model of the system under study. The experiments definition process is presented in Section 5.3. After that, Section 5.4 details the control flow for the execution of experiments and the particularities of the implementation of the fault injection and deletion approaches for each of the considered fault models. How these results can be analysed to assess the dependability of the modelled system is presented in Section 5.5. Finally, Section 5.6 summarises the basic features of this tool and depicts its control flow in great detail.

5.2 Architecture of FADES

The architecture of FADES [93] can be divided into hardware and software components. The hardware component mainly consists of an FPGA that is used to implement the system's model. The software module manages fault emulation experiments to assess the dependability of the system under study.

The selection of the most suitable FPGAs family could be critical for obtaining a flexible and high-performance fault emulation tool.

Any of the FPGAs families previously presented in Section 3.4 could be a good choice for implementing the system's model. This selection must be guided by performance issues related to Run-Time Reconfiguration approaches.

This methodology causes that, for each fault to be injected into the system, the FPGA must be reconfigured once to inject the fault and, when dealing with transient ones, once more to delete it some time later. The transfer of information to and from the programmable device to modify its configuration memory takes some time which may delay the execution of the workload on the system. This time evaluation was carried out in Chapter 4 for each proposed transient and permanent fault model. It must be minimised to improve the speed-up the fault emulation process may attain with respect to simulation-based fault injection techniques. Having this in mind, the reconfiguration capabilities of the selected FPGAs families (see Section 3.3.2) will determine the global temporal cost of emulating a fault.

If the family of FPGAs only supports a global reconfiguration approach then the *whole* FPGA's configuration memory must be rewritten each time the system must be reconfigured. In this case, the delay the reconfiguration process introduces only depends on the size of the selected device rather than on the injected fault model.

When the FPGAs family features local reconfiguration capabilities, only a fraction of the FPGA's configuration memory must be modified. That portion of the configuration memory is in charge of changing the functionality of the system according to the occurrence of the desired fault. In this case, the delay introduced by the reconfiguration process matches that presented in previous chapters and is, obviously, smaller than the caused by the full reconfiguration of the programmable device.

Therefore, in order to improve the achievable speed-up with respect to simulation-based fault injection experiments, a family of FPGAs featuring local (also known as *partial*) reconfiguration capabilities should be chosen. Currently, partial reconfiguration is supported by two of the main families of FPGAs presented in Section 3.4, the Xilinx's Virtex [72] and the Atmel's AT40K [82] families.

The AT40KAL family is aimed at high-speed (DSP/processor-based) designs and can implement a variety of computation intensive, arithmetic functions. This family allows for the partial reconfiguration of FPGAs by following a windowing mechanism [94]. This a very flexible approach since small portions of the configuration memory can be modified to alter the functional behaviour of the system.

The Atmel's Integrated Development System (IDS) [95] supports the design of partial reconfigurable applications. As shown in [96] this tool allows the user to implement two different designs (A and B) and generate a reconfiguration file containing the differences between them. This file can be used to modify the contents of the FPGA's configuration memory to alter the behaviour of the system from A to B.

The main problem with this approach is that the user must implement, by means of a graphical place and route tool, all the different functionalities required by the system. In some cases, the structure of the new (faulty) design may be known beforehand such as, for example, when opening a line of the circuit. However, the manual implementation of all the possible open-line cases could be a very long and tiring process. Most of the time, the structure of the new system is unknown and depends on the current state of the system and not on its structure, such as when inverting the current logic value of a FF.

Hence, this approach could be useful in the development of partial reconfigurable applications but it is not very well-suited for emulating the occurrence of faults into the system.

The Virtex family delivers high-performance, high-capacity programmable logic solutions. The partial reconfiguration of this family of FPGAs is performed via its *SelectMAP* interface [97] on a column basis. Bits of the FPGA's configuration memory are grouped into one-bit wide vertical *frames* that expand across the whole device [98]. Those frames are the smallest portions of the configuration memory that can be read or write and hence, although flexible enough, somehow limit the attainable speed-up by the use of a partial reconfiguration approach.

The design of reconfigurable applications is supported, on one hand, by the Xilinx's Integrated Software Environment (ISE) [99] and PlanAhead [100] tools. This design methodology presents the same drawbacks as the Atmel's one, the user must know beforehand the structure of the system to implement, usually tediously, the desired functionalities of the system and obtain the reconfiguration files that will modify its behaviour. Although it has been successfully used for the design of partial reconfigurable applications, it is not qualified for fault emulation.

On the other hand, Xilinx also provides a Java package, named JBits, which conforms an Application Programming Interface (API) to access and manage the configuration file of Virtex FPGAs. The origin of this API was JERC (***J**ava **E**nvironment for **R**econfigurable **C**omputing*) [101], a set of Java classes that allows the user to specify the structure of the desired circuit as if developing an application using the Java programming language. The compilation of this program generates the configuration file (*bitstream* or *.bit* in Xilinx's terminology) to be downloaded to the configuration memory of the FPGA. JBits extended this approach to give access to all the configurable features of Virtex FPGAs [102]. As shown in Figure 5.1 the user's program can supply and retrieve configuration control and data information to and from the reconfigurable logic. The interface to the hardware is provided by XHWIF (***X**ilinx standard **H**ard**W**are **I**nter**F**ace*).

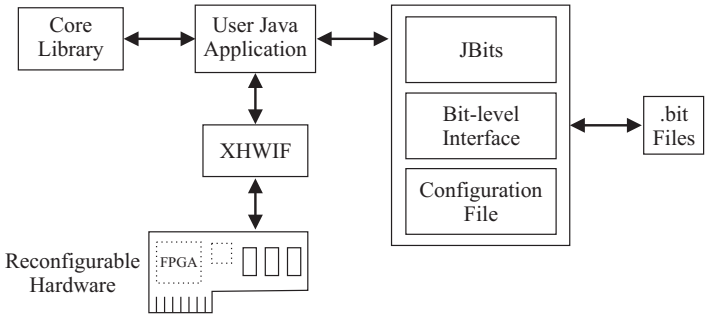

Figure 5.1: *JBits design flow.*

Another extension that was later integrated into JBits was JRTR [103], a Java package that provided the means to manage and generate configuration files for the partial reconfiguration of Virtex FPGAs. Nowadays, JBits is a faster approach than existing development tools for partial reconfigurable applications, although a deep knowledge of the internal architecture of the FPGA is required to use it.

JBits seems then a suitable tool for implementing a fault emulation tool based on Run-Time Reconfiguration techniques. The program developed by means of the JBits API may monitor the FPGA execution and, when necessary, can generate a new configuration files depending on the *current* state and structure of the FPGA. This file can be downloaded onto the configuration memory of the programmable device by using its partial configuration capabilities, thus minimising the number of bits to be transferred. That approach can be used to inject and delete a fault to and from the system when emulating its occurrence.

Hence, the family of FPGAs that has been selected to implement the first prototype of the fault emulation tool (FADES) is the Xilinx's Virtex. Despite that the minimum reconfiguration unit (frame) is larger than the windowing scheme from Atmel, Virtex FPGAs are larger than AT40K devices (hold a greater number of internal resources) and thus may implement more complex designs. Moreover, the JBits Java package is more flexible than Atmel's tools and can be efficiently used for dynamically generate partial reconfiguration files. JBits complexity will be hidden from the final user, who will only work with a Graphical User Interface (GUI) acting as an interface to the application developed using JBits.

According to this study, the final architecture of FADES can be seen in Figure 5.2.

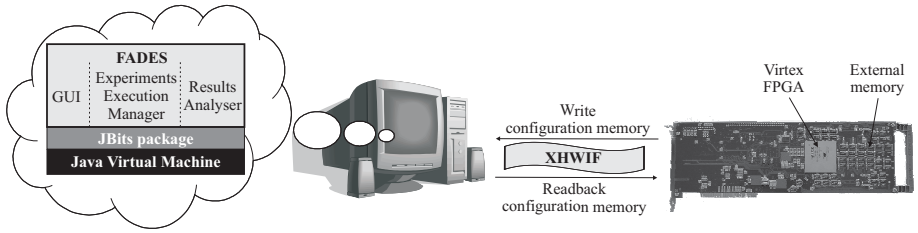

Figure 5.2: *Architecture of FADES.*

Any prototyping board holding a Virtex FPGA and featuring a XHWIF to communicate with a host can be used to implement the model of the system under study. In case that a XHWIF has not been specified for that particular board, it is possible to develop a custom one as explained in the *XHWIF Porting Guide* [104].

A host machine is needed to run the software components of FADES. These software modules have been entirely written in Java and, therefore, any platform with an available Java Virtual Machine is eligible for executing these applications. They make use of the JBits package to access the configuration memory of the FPGA to obtain information about the current state of the system, generate new configuration files containing the modifications required to change the behaviour of the system according to the fault injected, and download these new files onto the configuration memory of the programmable device. This application comprises several Java classes managing each one of the three identified key processes a fault emulation tool should consists of.

This software module comprises three different processes that manage the whole experimentation process.

- **Experiments definition.** A GUI supports the definition of fault injection experiments by selecting the bitstream file, the kind of faults to be injected, and the injection and observation points, among other things.

- **Experiments execution.** Once all the parameters have been defined, FADES takes control of the system to perform all the requested fault injection experiments. JBits methods are used to modify the contents of the configuration memory of the FPGA to inject/delete the fault into/from the system. The whole set of hardware fault models previously described in Chapter 4 has been implemented according to the Virtex architecture. The state of the system is recorded during the workload execution for each one of the experiments.

- **Analysis of results.** Faulty trace files are compared to a fault free execution (Golden Run) trace to determine the behaviour of the system in the presence of the injected faults.

Next sections describe how FADES manages and implements each one of these processes.

5.3 Experiments definition

This process tries to gather together all the necessary information to perform the desired fault emulation experiments to assess the dependability of the system under study. As shown in Figure 5.8, this information is mainly collected from user inputs and from the system's model and its implementation.

5.3.1 Model implementation and basic information

FADES requires some data regarding the structure of the system under study and general aspects of the experiments, to be able to help the user to comfortably define fault emulation experiments. The user must provide this information by means of the graphical user interface (GUI) shown in Figure 5.3.

Most of these data are provided by means of a number of files:

a) **Bitstream file**. This is the result of synthesising and implementing the model of the system onto the selected FPGA (from the Virtex family, in this case) following the common design flow described in Section 3.2.

   Although any commercial synthesiser can be used to extract from the model the basic described components and their interconnection, the place and route tools are constrained to those provided by the FPGA

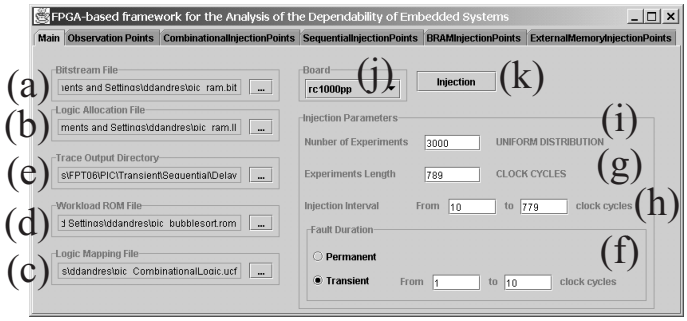

Figure 5.3: *FADES GUI: Form to retrieve basic information.*

manufacturer. In this case, the Xilinx's ISE tool must be used to distribute the components among the unused FPGA's internal resources and generate the bitstream file.

This is the file that will be downloaded onto the FPGA's configuration memory to emulate the fault-free behaviour of the system. As it contains all the information related to the structure of the implemented system, it can be analysed by means of JBits to extract such information. The structure of this file is detailed in Appendix A.1.

b) **Logic Allocation file**. The Logic Allocation file (.ll), which can be generated by properly instructing the ISE tool, exposes in a proprietary format the position of all the sequential elements (flip-flops and memory bits) used to implement the system's model in the FPGA's array of configurable blocks. Each sequential element is also related to its name in the original model of the system. Appendix A.2 details the structure of this file.

   This information can be extracted later by parsing the file to obtain the location of all these elements, which will be useful to determine the sequential injection points and/or the observation points.

c) **User Constraints file**. The User Constraints File (.ucf) contains, in a proprietary format, information destined to guide the implementation of the system's model, such as temporal or allocation constraints.

   However, the Floorplanner tool of the ISE framework may be used to generate a User Constraints File (.ucf) containing the *final* position of all the logic elements of the system. The structure of this file is shown in Appendix A.3.

   Information extracted from this file may be of use to determine the available combinational injection points.

d) **Workload file**. A simply way of providing the workload to the implemented system is via the FPGA's internal memory or the external memory of the prototyping board.

   FADES may obtain this information by parsing the specified workload file. The custom format of this file is presented in Appendix A.4.

e) **Trace output directory**. Files containing the trace of the workload execution will be stored in this directory after the experiment's conclusion.

The rest of parameters are grouped in the form shown in Figure 5.3.

f) **Fault's duration**. Permanent faults will remain in the system until the end of the experiment, whereas transient ones will disappear after the fault duration. Currently, the duration of each transient fault (in clock cycles) is uniformly distributed between the lower and upper thresholds defined by the user.

g) **Experiment's duration**. The length of the experiments is defined in terms of clock cycles, which makes it independent of the clock frequency.

h) **Fault injection interval**. It defines the temporal upper and lower thresholds in which faults may occur. The injection time for each experiment is uniformly distributed between these limits.

i) **Number of experiments**. At present FADES only supports the injection of single faults, and then the number of experiments also defines the number of faults to be injected.

j) **Board selection**. It allows the selection of the prototyping board holding the FPGA that will implement the model of the system. Currently, FADES natively supports the use of the RC1000 board [105] from Celoxica[1] and the XSV800 board from Xess[2].

k) **Proceed to experiments execution**. This button will trigger the experiments execution process.

5.3.2 Injection into sequential logic

The definition of where to inject faults into the sequential logic of the system's model, and which kind of faults must be considered, is assisted by the GUI depicted in Figure 5.4.

[1] http://www.celoxica.com/
[2] http://www.xess.com/

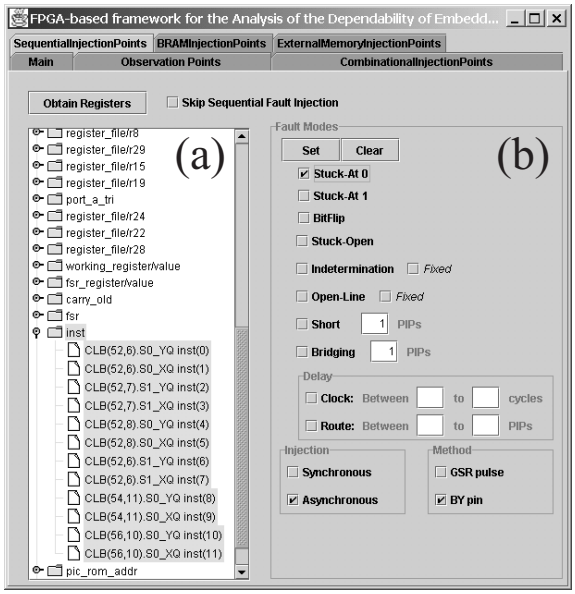

Figure 5.4: *FADES GUI: Form to define the location and type of faults affecting the sequential logic of the system.*

The most efficient way of obtaining the location of the whole sequential elements implemented in the FPGA is by parsing the Logic Allocation file generated by the ISE framework.

In case this file is not provided, it is also possible to search the bitstream file for FFs whose output is connected to some other elements. Although this can be done by using the JBits package, it is quite difficult to correlate the sequential elements found with its original name in the system's model. So, the use of the Logic Allocation file is greatly encouraged.

The window form shown in Figure 5.4a provides the information extracted from this analysis for the user to select the locations that might be affected by the occurrence of a fault. Faults will be uniformly distributed among them.

A very simple form, shown in Figure 5.4b, allows the user to select all the fault models that will be injected during the experimentation (one fault per experiment). It includes permanent and transient fault models affecting the sequential logic of the system, such as *stuck-ats*, *bit-flips*, *stuck-opens*, *indeterminations*, *open-lines*, *shorts*, *bridgings*, and *delays*. Different options for some of the faults may also be selected, such as the approach to be followed to inject the fault.

5.3.3 Injection into combinational logic

A similar GUI, illustrated in Figure 5.5, is used to specify the location of faults affecting the combinational logic of the system's model, and the kind of faults to be considered.

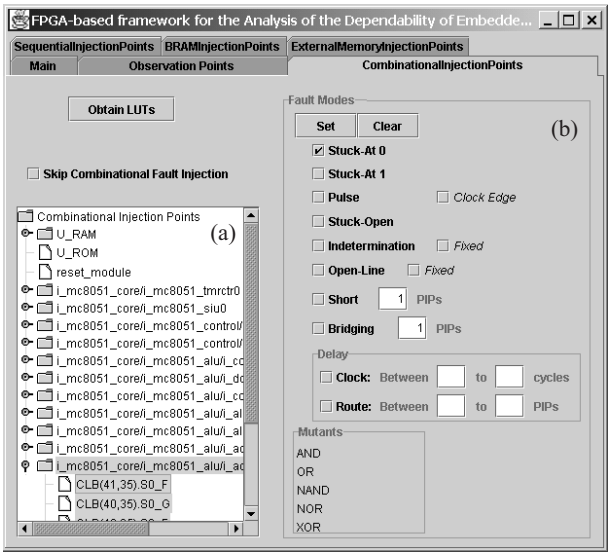

Figure 5.5: *FADES GUI: Form to define the location and type of faults affecting the combinational logic of the system.*

Although information related to the sequential elements of the implemented system may be obtained quite straightforwardly, information about the combinational elements of the system is more complex to get.

JBits classes and methods can be used to determine whether each LUT in the FPGA has been used to implement combinational logic of the system's model or not.

This can be done by scanning the bitstream file to check whether a particular LUT is driving any of the possible CB outputs and whether this output is driving any other element of the system. If this is the case, the contents of the LUT are retrieved for later use. Table 5.1 describes a simplified version of the function implemented to discover all the LUTs used in an implemented system's model.

This procedure is very useful to determine the contents and location all the LUTs used in the final implementation, but it does not provide any information regarding the relation between these LUTs and the related combinational logic of the system's model.

This problem can be solved by matching the LUTs location with informa-

Table 5.1: *Sample code for determining whether the LUTs* G *located in slice* 0 *are being used and get their contents.*

```
// Check all the LUTs of the FPGA
for (int row = 0; row < clbRows; row++) {
  for (int col = 0; col < clbColumns; col++) {
    outBUsed = false; outQUsed = false; outUsed = false;

    // Check whether the LUT and outB are in use
    // Check whether the LUT controls the SOControl.YCarrySelect.YCarrySelect multiplexer
    control = jbits.get(row, col, SOControl.YCarrySelect.YCarrySelect);
    if (control == SOControl.YCarrySelect.LUT_CONTROL) {
      // Obtain the routing tree starting from outB (YB)
      source = new Pin(Pin.CLB, row, col, CenterWires.SO_YB);
      rtree = new RouteTree(source);
      tracer.trace(rtree);
      //outB drives some element if its routing tree has some children
      if (rtree.numChildren() > 0) used = true;
    }

    // Check whether the LUT and outQ are in use
    // Check whether the LUT drives the SOControl.YDin.YDin multiplexer
    control = jbits.get(row, col, SOControl.YDin.YDin);
    if (control == SOControl.YDin.Y) {
      // Obtain the routing tree starting from outQ (YQ)
      source = new Pin(Pin.CLB, row, col, CenterWires.SO_YQ);
      rtree = new RouteTree(source);
      tracer.trace(rtree);
      //outQ drives some element if its routing tree has some children
      if (rtree.numChildren() > 0) used = true;
    }

    // Check whether the LUT and out are in use
    // Obtain the routing tree starting from out (Y)
    source = new Pin(Pin.CLB, row, col, CenterWires.SO_Y);
    rtree = new RouteTree(source);
    tracer.trace(rtree);
    //out drives some element if its routing tree has some children
    if (rtree.numChildren() > 0) used = true;

    if (outBUsed || outQUsed || outUsed) luts[row][column][0][0] = jbits.get(row, col, LUT.SLICE0_G);
  }
}
```

tion extracted from the User Constraints file.

Finally, LUTs are grouped (see Figure 5.5a) according to the system's components they belong to. The user can then select those locations that will be affected by faults occurring in the combinational logic of the system's model.

The kind of faults that will be considered for experimentation is selected by using the form shown in Figure 5.5b. Available faults include *stuck-ats*, *pulses*, *stuck-opens*, *indeterminations*, *open-lines*, *shorts*, *bridgings*, and *delays*. Several options, such as the approach to be followed to inject the fault, can be also selected for some of them.

5.3.4 Monitoring

The system's workload execution in the presence of faults must be monitored to determine the effects of faults on the system's behaviour.

As previously explained FPGAs are synchronous devices destined to implement synchronous systems. There is not a straightforward way of obtaining the current state of a combinational signal at a particular time. However, accessing the configuration memory of the device allows for retrieving both the system's structure and the state of all the synchronous elements of the FPGA.

Thus, only sequential elements may be monitored.

Figure 5.6 depicts the GUI that is used to specify those elements of the system (observation points) that must be monitored.

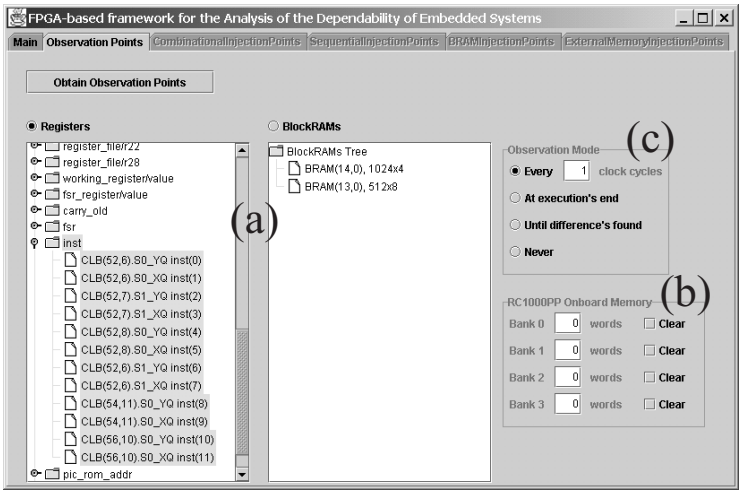

Figure 5.6: *FADES GUI: Form to define the observation points to monitor the execution of the system's workload.*

The location of all the sequential elements of the system's model after its implementation can be extracted, as previously explained, by parsing the Logic Allocation file. This information is exposed (see Figure 5.6a) for the user to easily select those elements that will be monitored. The external memory of the prototyping board (cf. Figure 5.6b) may also be considered for observation.

Finally, the user can select among different monitoring policies (through the form shown in Figure 5.6c). According to it, the system will be monitored i) every number of clock cycles, ii) until a difference with the fault-free execution of the system (Golden Run) is found, iii) only at the end of each experiment, or iv) never at all. Each of these policies is intended to be useful for different purposes.

5.3.5 Summary

FADES' GUI can be used to collect all the files and user inputs required to define the experiments to be carried out in a fault injection campaign.

Information related to the implementation of the system's model is presented to the user in a friendly manner to support the definition of the experiments. Apart from generic information regarding the experiments, the user must provide the selected fault injection points and observation points.

Once all these parameters are defined, FADES assumes the control of the system for experimentation.

5.4 Experiment execution

This process is in charge of controlling the entire experimentation flow. Figure 5.8 depicts all the procedures this flow involves, including the initialisation of the prototyping system, the execution of the workload, the observation of the system's behaviour, the injection and deletion of faults, and the reset of the system to its initial state.

All these procedures, which make use of the JBits package to easily access the configuration memory of the programmable device, are next detailed.

5.4.1 Initialisation

Before executing any experiment, the FPGA must be programmed with the bitstream file obtained in the experiment's definition phase. The configuration data of the bitstream is downloaded into the configuration memory of the FPGA, changing its functional behaviour as defined by the model of the system. A simplified version of this initialisation code is described in Table 5.2.

This initialisation step may also program the board's and/or FPGA's memory blocks with the required workload. Data from the workload file, which is firstly parsed to translate it into a JBits compatible format, is downloaded into the selected memory blocks. This procedure is detailed in Table 5.3.

Now that the system is completely initialised FADES may successively launch each one of the defined fault injection experiments.

Table 5.2: *Sample code for initialising the prototyping board and configuring the FPGA with a bitstream file.*

```
// Get a XHWIF interface to the board (rc1000pp in this case)
board = XHWIF.Get(boardName);
// Get a remote host name to access the board from the network
remoteHostName = XHWIF.GetRemoteHostName(boardName);
// Get a remote port number, if any
port = XHWIF.GetPort(boardName);

// Establish a connection to the board
board.connect(remoteHostName, port);
// Reset the board
board.reset();
// Stop the clock
board.clockOff();
// Get the devices available on the board (a XCV1000 FPGA)
deviceType = board.getDeviceType();

// Create a JBits object from the first/only device on the board
jbits = new JBits(deviceType[0]);
// Read in the bitstream file
jbits.readPartial(inBitFileName);
// Get the data to be downloaded onto the FPGA
partial = jbits.getPartial();

// Trigger the GlobalSetReset (GSR) after each reconfiguration
jbits.enableSoftReset(true);
// Download the configuration data to the FPGA (device 0)
board.setConfiguration(0, partial);
// Do not trigger the GSR after each reconfiguration
jbits.enableSoftReset(false);
```

Table 5.3: *Sample code for programming an external memory bank of the prototyping board with the information contained in a workload file.*

```
// Get a new WorkloadParser object for the workload file
workloadParser = new WorkloadParser(workloadName);
// Parse the file and extract the contents of the memory block
rom = workloadParser.getMemoryContent();

/* JBITS FUNCTION */
// Write the data into the board's memory bank 1 (address 0x200000)
board.setRAM(address, rom);
```

5.4.2 Workload execution

The first problem that arises when trying to carry out the experiments is related to the temporal management of the experiments.

Faults must be injected at a particular time, they may be deleted some time afterwards, and even experiments last a certain amount of time. As the board's clock continuously generates pulses at a defined frequency, there is no way to control in which moment a particular event should happen in the system. However, the JBits *XHWIF* interface, provides some methods for managing that clock.

The method *clockOff()*, already presented in Table 5.2, may be used to turn off the board's clock, thus preventing the clock line to be continuously pulsed. Now that the board's clock is ignored, FADES can generate any desired number of clock pulses by calling the software method *clockStep(int count)*.

In this simple way, the system's workload may be executed any number of clock cycles, from a single one to the whole experiment's length. Hence, it allows for holding the current state of the system, performing the desired reconfigurations, and resuming the system's execution afterwards. Furthermore, this procedure will support the fault injection into systems were on-the-fly reconfiguration is not available, or when the reconfiguration time exceeds the system's clock period.

So, as Figure 5.8 shows, the system's clock is first stepped a number of cycles equal to the experiment's injection time. At this point in time, the selected fault must be injected at the specified location.

After that, the system's clock is stepped until either the fault disappears from the system (transient fault), or the experiment's end is reached (permanent fault). In any case, the fault must be deleted from the system, either to continue the execution of the workload until the end of the experiment (transient fault), or to allow for the execution of a new experiment with a fault-free configuration (permanent fault).

When dealing with transient faults, the system's clock must be stepped again the number of clock cycles required to reach the end of the experiment.

It is to note that any fault injection campaign must performed, at least, one fault-free execution of the system's workload, named *Golden Run*. The Golden Run's execution is monitored for an oracle to later determine the effects of the injected faults on the system's behaviour.

5.4.3 Observation points

The system must be monitored during the execution of the experiments to later analyse the effects of the injected faults on the system.

According to Section 5.3.4, the points of interest to be observed (observation points) must be selected among the sequential elements of the system under study (FFs and memories). The current state of all these elements is stored in the related bits of the configuration memory of the FPGA.

The observation rate defined by the user determines when the state of these observation points must be retrieved from the FPGA's configuration memory.

The JBits package is again very helpful to monitor the observation points of the system, i.e. obtaining the current state of the selected sequential elements. The *ReadbackCommand* class provides methods for getting data back from the FPGA through a *readback* command. The structure of this command is detailed in Appendix A.5. Table 5.4 describes in a simplified way the procedure to be followed to obtain the current state of a certain FF.

The information extracted in this way is stored in a trace file for later analysis. The particular structure of this file is described in Appendix A.6.

Table 5.4: *Sample code for retrieving the current state of FF* X *from slice* 0 *located in row* row *and column* column.

```
// Create a state readback interface to JBits
state = new State(1);
// Assign a bit in the state vector to the observation point
state.setBitEntry(0, row, column, CLB.SLICE0_XQ);

// Generate the readback command
readCommand = ReadbackCommand.getClbRegs(deviceType[0], column,
                                         ReadbackCommand.SLICE0_XQ);
// Get the readback data's length in bytes
readbackBytes = (ReadbackCommand.getReadLength() * 4);

// Send the readback command
result = board.setConfiguration(0, readCommand);

// Readback the expected data
data = board.getConfiguration(0, readbackBytes);
// Parse the retrieved data
jbits.parsePartial(readCommand, data);
// Retrieve the current bit value of the observation point
currentState = (state.getIntArray(jbits))[0];
```

5.4.4 Fault injection

The fault injection procedure involves processing all the available information extracted from the bitstream, and possibly request some information from the FPGA like the current state of some FFs, to compute the reconfigurable file that emulates the behaviour of the system in the presence of the desired fault.

JBits features a class, also called *JBits*, which defines a programming interface to access all the internal resources of Xilinx's Virtex devices. The *get(int clbRow, int clbColumn, int[][][] bits)* method can be used to get the current configuration of a particular component, determined by the *bits* parameter, and located at the CB in row *clbRow* and column *clbColumn*. Likewise, the *set(int clbRow, int clbColumn, int[][][] bits, int[] val)* method allows for modifying the current configuration of the selected component to the one specified by the *val* parameter. These two methods operate on the JBits object only, and do not update the current FPGA's configuration.

To do so, the data of this new configuration must be downloaded onto the FPGA's configuration memory. In order to reduce the amount of data to be transferred, and thus the temporal overhead of the reconfiguration process, it is possible to take advantage of the local (partial) reconfiguration capabilities of Virtex FPGAs. The *getPartial()* method returns a partial configuration bitstream containing only those bits (actually those frames) the current configuration of the FPGA differs from the JBits object.

The transfer of this bitstream causes the FPGA to modify the configuration of its programmable components to behave as the system in the presence of the desired fault. Table 5.5 describes a simplified version of this algorithm.

Table 5.5: *Sample code for injecting a* stuck-at-0 *fault into the output of the LUT* G *of slice* 0 *located at the CB in row* row *and column* column.

```
// Step the clock until the injection time is reached
board.clockStep(injectionTime);
// Get the contents of the targeted LUT before injecting the fault
faultFreeLUT = jbits.get(row, column, LUT.SLICE0_G);

/* FAULT INJECTION*/
// Get the truth table of the faulty LUT
faultyLUT = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
// Modify the JBits object to reflect the desired changes
jbits.set(row, column, LUT.SLICE0_G, Util.InvertIntArray(faultyLUT));
// Generate the partial configuration file containing these changes
partialBitstream = jbits.getPartial();
// Download the partial bitstream to device 0
board.setConfiguration(0, partialBitstream);
```

As reported in Chapter 4, different approaches should be followed for injecting each one of the considered hardware faults. These approaches, that have been defined for a generic FPGA architecture, must be now adapted for the particular architecture of Virtex FPGAs. JBits classes and methods will be used, as previously explained, for this purpose.

For simplicity, just a summary of those differences that are worth mentioning follows.

- **Bit-flip**. The main issue in this fault model is related to the use of the LocalSetReset (LSR) signal approach.

  Virtex CBs consists of two slices that hold two FFs each. Both FFs share the lines that control their set/reset logic. Therefore, pulsing the LSR line will cause both FFs to be set or reset according to the configuration of the related logic.

  Hence, the injection of a *bit-flip* in one of the FFs will involve i) reading the current state of both FFs, ii) configuring the control logic to invert the current value of the targeted FF and keep the state of the other, and iii) pulsing the LSRin line as explained in Section 4.2.2.

- **Stuck-at**. As in the case of *bit-flips*, the use of the LSRin line to change and keep the state of a targeted FF will also affect the other FF that is sharing that line. Therefore, this approach may only be applied when only one of the FFs of a CB is being used for implementing the system's model.

  The same problem appears when using the CLKin-based approach, since FFs within the same CB also share the clock line.

  Hence, although as shown in Section 4.6, the injection of *indeterminations* into sequential logic by using the LUT-based approach incurs the highest temporal overhead, it could be the only available possibility in some cases.

- **Pulse**. The implementation of this fault model is quite straightforward with the aid of the JBits package and do not present many problems. However, problems arise when the duration of *pulses* injected into a LUT extends beyond the next clock cycle.

  Virtex FPGAs do not provide any means to obtain the current state of combinational lines. The proposed approach, which consists in inverting the current state of the targeted logic by including a NOT gate into the circuit, works well for faults that disappear after the next clock cycle. The problem of further extending the duration of the fault is that the

pulse must invert and *maintain* the state of the logic element for the duration of the fault, which may not be achieved by just a NOT gate.

The only way of solving that particular problem consists in executing two different experiments: one considering that the *pulse* must keep a `'0'` and the other a `'1'`. This can be modelled as a *transient stuck-at 0* and a *transient stuck-at 1*. FADES will be able to determine the result of the *pulse* fault by comparing the results of both *transient stuck-at* faults.

Hence, this approach will double the temporal cost estimation for injecting a *pulse*.

- **Stuck-open**. As in the case of *pulses*, the main issue when dealing with *stuck-opens* in combinational logic is that the current state of the targeted combinational element cannot be obtained. Hence, it is not possible to keep the current state of the logic since it is unknown.

  To solve this problem, two experiments are performed, assuming that the element holds a low- and a high-logic level respectively. Results from both experiments are compared to determine the result of the *stuck-open* fault. Of course, this will double the expected temporal cost of injecting this fault.

  Another problem is keeping constant the state of a targeted FF. Depending on the system some of the faster approaches may not be used, as in the case of *indeterminations* and *stuck-ats*.

- **Indetermination**. The implementation of this particular fault presents the same problems already discussed for *stuck-ats*.

- **Delay**. The main problem with this fault model is not its implementation but the JBits routing algorithms from the *JRoute2* class [106].

  On one hand, the memory requirements for using the provided methods are huge. On the other hand, the routing algorithms, although very helpful for testing purposes, present serious problems when dealing with congested designs.

  As the purpose of this work is not designing new and complex routing techniques for FPGAs, we have decided to make use of JRoute2 as it is. This means that, depending on the design, it could not be possible to inject a delay with the desired propagation delay but a smaller one.

- **Short, Open-line, and Bridging**. The implementation of all these faults does not present very complex problems. As they are mainly routing related tasks, the use of JBits routing facilities greatly eases the process of modifying the routing of the system.

5.4.5 Fault deletion

Transient faults have a duration after which they must be deleted from the system by following the steps presented in Section 4.11.

Permanent faults remain into the system and, thus, their duration is limited by the duration of the experiment. At the end of the experiments the permanent fault must be deleted to allow for the execution of new experiments with a fault-free implementation of the system. Therefore, the operations described in Tables 3.1 and 3.2 may be used again to restore the original fault-free configuration of the system. As this approach takes benefit of the partial reconfiguration capabilities of Virtex FPGAs, it is faster than downloading the whole original bitstream file to remove the fault.

The fault deletion process for each particular permanent fault is next summarised in Table 5.6 following a C-like pseudo-code. This process complements the one described in Tables 4.4, 4.8, 4.9, and 4.10, for emulating the occurrence of permanent faults.

Table 5.6: *Pseudo-code for deleting a permanent fault from the system at the end of the experiment.*

Fault model	Fault deletion
Stuck-at	`deleteTransientIndetermination();`
Stuck-open	`deleteStuckAt('0');`
Indetermination	`deleteTransientIndetermination();`
Delay	`deleteTransientDelay();`
Short	`for (i = 0; i < newRouting.length; i++)` `  ((PT) newRouting[i])(off);`
Open-line	`target(on);`

Bridging	`target(on);`
	`for (i = 0; i < newRouting.length; i++)`
	`  ((PT) newRouting[i])(off);`

FADES makes use again of JBits methods to generate the configuration file required to restore the fault-free configuration of the system. The same flow previously presented for injecting the fault is now used to delete it.

A simplified sample code for deleting a previously injected *stuck-at-0* fault is described in Table 5.7.

In case of dealing with permanent faults the end of the experiment has already been reached. Otherwise, the system's clock must be stepped the remaining number of clock cycles.

Table 5.7: *Sample code for deleting a* stuck-at-0 *fault injected into the output of the LUT* G *of slice* 0 *located at the CB in row* row *and column* column.

```
// Step the clock until the fault elapses
board.clockStep(faultDuration);

/* FAULT DELETION */
// Modify the JBits object to reflect the desired changes
jbits.set(row, column, LUT.SLICE0_G, Util.InvertIntArray(faultFreeLUT));
// Generate the partial configuration file containing these changes
partialBitstream = jbits.getPartial();
// Download the partial bitstream to device 0
board.setConfiguration(0, partialBitstream);
```

5.4.6 Reset to initial state

Once the experiment ends, the system is reset to its initial state, as shown in Table 5.8, to allow for the execution of another experiment from scratch.

Table 5.8: *Sample code for reseting the FPGA at the end of an experiment.*

```
// Step the clock until the end of the experiment is reached
if (transientFault) jbits.stepClock(remainingTime);

/* RESET TO INITIAL STATE */
// Trigger the GlobalSetReset (GSR) after each reconfiguration
jbits.enableSoftReset(true);
// Download the configuration data to the FPGA, resetting the device
board.setConfiguration(0, null);
// Do not trigger the GSR after each reconfiguration
jbits.enableSoftReset(false);
```

5.5 Analysis of results

Measurements performed during the workload's execution, i.e., the state of the observation points at that particular time (see Table A.7), are stored in a trace file for each experiment carried out. At the end of the experimentation, those traces must be analysed to determine the effects of faults on the system. The GUI depicted in Figure 5.7 helps the user to configure this analysis.

First of all, the user must indicate the directory where all the traces are located by means of the form shown in Figure 5.7a.

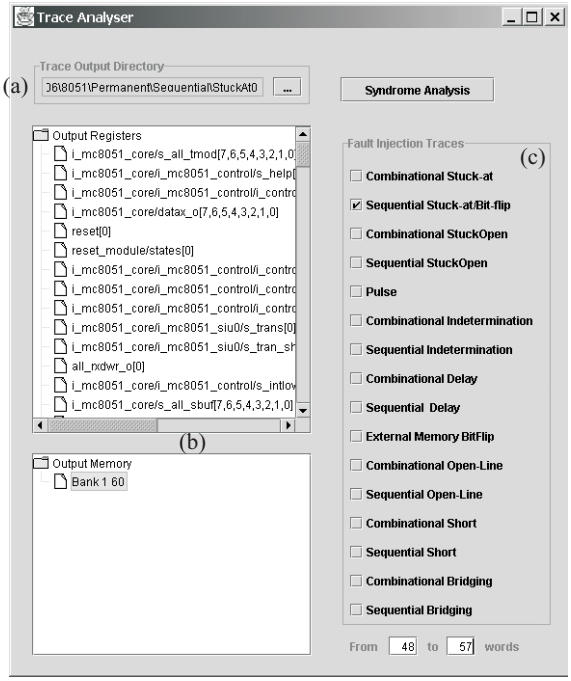

Figure 5.7: *FADES GUI: Form to configure the analysis of the fault injection experiments' traces.*

This information is parsed and provided to the user, so he can determine those observation points considered more significant for the analysis of the system's behaviour (cf. Figure 5.7b). In this way, for instance, it is possible to monitor all those observation points that constitute the *state* of the system, and select as key points those related to system outputs.

It is also possible to configure the analyser to extract information related to some particular injected faults (see Figure 5.7c).

The Golden Run and the faulty traces files are parsed, and the state of the observation points is compared to determine whether the fault affected the normal execution of the workload.

This comparison provides three kind of measures for each observation point.

- In first place, it obtains the number of experiments in which the traces of the Golden Run and the faulty experiment are identical. This obviously means that the system has, somehow, tolerated the fault.

- Next, it presents the number of experiments in which observations from the faulty trace do not match those of the Golden Run, but they are not

included in the range of observation points selected as most significant. The system is moderately affected by the fault occurrence.

- Finally, it computes the number of experiments in which observations considered as most significant do not match those of the Golden Run. The system is severely affected by the occurrence of the fault.

Results are not only provided as absolute numbers but also as percentage of experiments. Global numbers for the whole experimentation are also included.

All this information is stored in a text file, whose structure is described in Appendix A.7.

5.6 Conclusions

FADES (***F****PGA-based Framework for the* ***A****ssessment of the* ***D****ependability of* ***E****mbedded* ***S****ystems*) is the first prototype of an FPGA-based fault injection tool that follows a Run-Time Reconfiguration approach for the fast and early emulation of the behaviour of a system in the presence of faults.

The tool's hardware module features a Xilinx's Virtex FPGA, which enables the partial reconfiguration of the programmable device and allows for the use of the JBits package. This Java package provides some useful classes and methods for retrieving information from a Virtex FPGA and generating new configuration files that can be downloaded onto the FPGA's configuration memory. The communication to and from the FPGA is controlled by means of a host platform that also is in charge of running the software module.

The global control flow of FADES is depicted in Figure 5.8.

FADES copes with all the previously identified goals for this tool: i) it allows for the early and fast dependability assessment of system's models following a Run-Time reconfiguration approach, ii) it implements the whole set of hardware fault models considered representative of deep-submicron manufactured systems, and iii) it presents a very simple GUI that any non-skilled user may employ for evaluating the dependability of his systems.

This shows the feasibility of all the proposed approaches for the injection of faults by means of FPGAs. It only rest to determine the advantages and limitations of this first prototype by means of experimentation.

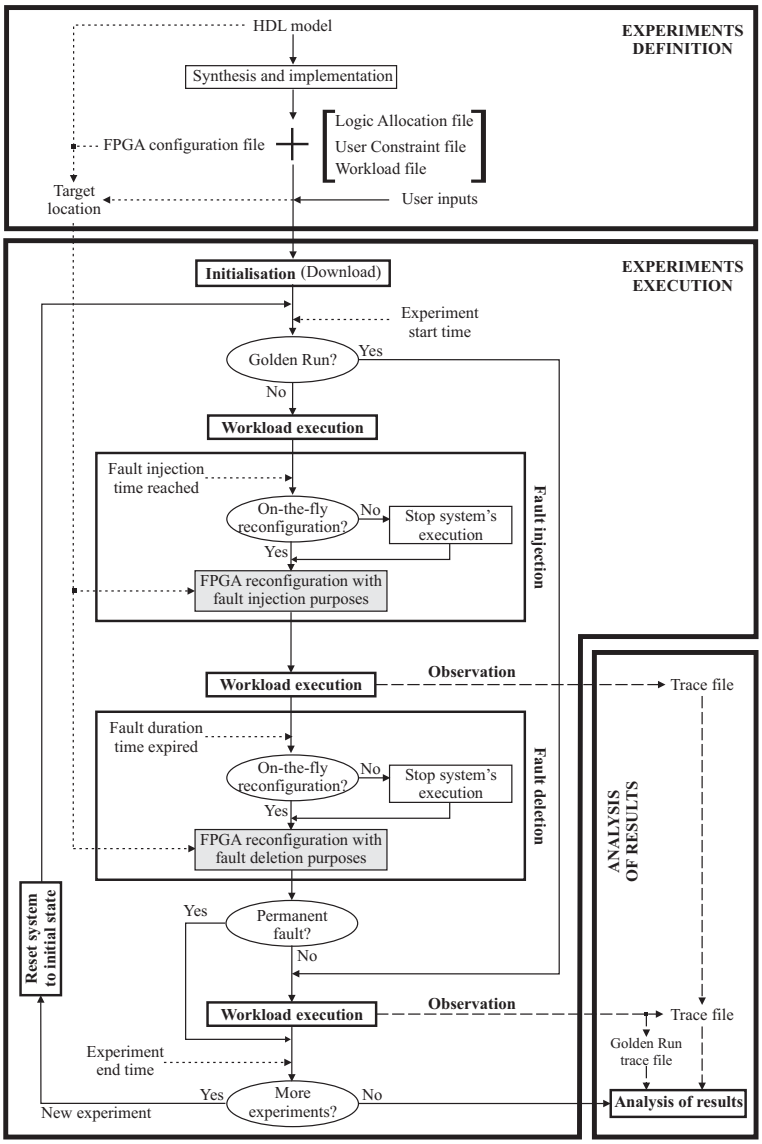

Figure 5.8: *FADES: fault injection campaign control flow.*

Chapter 6

Experimental Validation and Case Study

FADES implements the proposed methodology for emulating a wide range of hardware faults following a Run-Time Reconfiguration approach. That tool is intended to accelerate simulation-based fault injection experiments while correctly emulating the behaviour of the system in the presence of faults.

This Chapter presents the experimental procedure followed to validate the correctness of the results provided by FADES and the attainable speed-up when comparing the experimentation time with that provided by a state of the art simulation-based fault injection tool. The advantages and drawbacks of this implementation are also discussed along with a case study that shows the usefulness of this tool for the dependability assessment of embedded systems.

6.1 Introduction

Although the aim of FADES is speeding-up simulation-based fault injection experiments, it is very important to keep the focus on correctly emulating the system's behaviour in the presence of faults. Hence, the main goals of this Chapter are i) validating the proper behaviour of emulation experiments conducted by means of FADES and, ii) determining the attainable acceleration achieved by using this tool instead of simulation-based techniques. Different experiments have been performed to obtain significant results that can be use to determine whether these goals have been achieved.

Furthermore, these results also allow for the discussion of other interesting points, like iii) the advantages and drawbacks derived from the current implementation of this tool, and iv) the possible application of FADES as a tool for assisting the dependability evaluation of systems.

Similar experiments have been carried out by FADES and VFIT (***V**HDL-based **F**ault **I**njection **T**ool*) [107], a state-of-the-art simulation-based fault injection tool. Section 6.2 presents the validation of FADES' results by comparison with those provided by VFIT. The execution speed-up that FADES achieves with respect to VFIT is reported in Section 6.3. Section 6.4 discusses different applicability-related features of this tool, while Section 6.5 presents different examples of the use of FADES to assess the dependability of microprocessor-based systems. Section 6.6 summarises the results of this study and draws the main conclusions of this work.

6.2 Correctness of experiments' results

Results provided by FADES consist in the percentage of experiments that affected the behaviour of system under study in three different degrees (not affected, moderately affected, severely affected), as defined in Section 5.5.

Determining the correctness of these results is not an easy task. It should be necessary to exactly know the behaviour of the system in the presence of that fault beforehand and trace the execution of the workload after the insertion of the fault, for an oracle to assure that the expected faulty behaviour is being accomplished. Obviously, making this study while taking into account the whole set of considered fault models and eligible injection points, even for a small system, is an unfeasible task.

The approach proposed here for estimating the correctness of the results provided by FADES, relies on comparing its results with those provided by a state-of-the-art simulation-based fault injection tool. This tool, named VFIT, makes use of the commercial ModelSim[1] simulator to introduce the faults into the model of the system. Faults can be injected during the execution of the system's workload, following a *simulator commands* approach [29], by modifying the current logical value of signals and variables of the model. The faults that can be injected by this technique include *bit-flips*, *pulses*, *stuck-ats*, *stuck-opens*, *indeterminations*, and *open-lines*.

6.2.1 Experimental set-up

A model of the Intel 8051 microcontroller provided by Oregano Systems [108] was selected for experimentation. This is not a large size model, which allows for following the trace of the system's execution relatively easily. In this way,

[1] http://www.model.com/

it could be possible to detect and correct any problem related to the current implementation of FADES.

A *bubblesort* algorithm, commonly used in the validation of other model-based fault injection tools, was selected as the workload of the system.

The aim of these experiments was to validate the correctness of the results provided by FADES and the current implementation of all the considered hardware faults. Some preliminary experiments were performed to determine those points of the system's model in which the occurrence of a fault most likely affect the proper behaviour of the system. This information was used for the selection of fault injection points to increase the effectiveness of the injected faults. In this way, it could be easier to compare results from both tools as a significant number of experiments will result in a wrong behaviour, which could not be ascertained otherwise.

This exploratory operation consisted in 3000 single *bit-flip* and *pulse* injected into the sequential and combinational logic of the system using FADES. These faults were uniformly distributed along the duration of the workload and the eligible injection points.

As a result, the registers to be targeted by faults affecting the sequential logic of the system were the *datax_o*, *psw* (control unit), *pc*, *s_preadr_2*, *psw* (memory unit), *state*, *s_r1_b0*, *s_r0_b0*, *s_intpre2*, *s_ir*, *sram_addr*, *s_ram_data_out*, *Mtrien_sram_data*, *s_rom_data*. Regarding the faults affecting the combinational logic of the system, the eligible injection points of the *Arithmetic-Logic Unit* were selected.

According to this information, a new fault injection campaign was set-up to execute 3000 experiments for each one of the considered hardware fault models targeting the combinational and sequential logic of the system. Single faults were again uniformly distributed along space and time. The duration of transient faults was also uniformly distributed into three different ranges: faults with a duration less than 1 clock cycle, those lasting between 1 and 10 clock cycles (transient faults with an unusual long duration), and faults during between 10 and 20 clock cycles (very long duration, with results tending to those of permanent faults). These intervals were selected according to the common use of VFIT in some other case studies [109] [110] [111] to ensure the propagation of the faults.

6.2.2 Analysis of results

The results of all these experiments, carried out by both FADES and VFIT, are presented in Table 6.1 (transient faults) and Table 6.2 (permanent faults).

As the system under study did not feature any fault detection or fault tolerance mechanisms, only results related to experiments severely affecting the behaviour of the system under study are reported on these tables.

Table 6.1: *Percentage of experiments whose behaviour was severely affected by transient faults injected by means of FADES and VFIT. The duration of the faults was divided into three different ranges: less than one clock cycle, between one and ten clock cycles, and between ten and twenty clock cycles.*

Transient fault	Targeted logic	FADES			VFIT		
		Fault duration in clock cycles					
		< 1	1-10	10-20	< 1	1-10	10-20
Bit-flip	Sequential	43.86			43.70		
Pulse	Combinational	0.06	3.13	8.86	1.36	3.53	7.43
Indetermination	Sequential	29.73	50.86	73.3	18.87	35.90	52.47
	Combinational	0.33	2.2	5.4	1.30	3.03	8.23
Delay	Sequential	2.53	14.36	24.4	-	-	-
	Combinational	0.43	1.72	3.23	-	-	-

Table 6.2: *Percentage of experiments whose behaviour was severely affected by permanent faults injected by means of FADES and VFIT.*

Permanent fault	FADES		VFIT	
	Targeted logic			
	Sequential	Combinational	Sequential	Combinational
Stuck-at-0	56.36	10.56	43.30	16.37
Stuck-at-1	84.8	12.83	72.03	24.80
Stuck-open	68.53	9.63	-	-
Indetermination	71.26	12.33	68.33	25.23
Delay	36.2	5.66	-	-
Short	20.95	15.52	-	-
Open-line	45.8	19.33	68.47	25.20
Bridging	25.8	14.06	-	-

At first glance, the results provided by both tools are consistent with the expected impact of permanent and transient faults into the sequential and combinational logic of a system.

On one hand, as permanent faults remain into the system, they lead to a higher rate of affected experiments than transient faults. The percentage of experiments whose behaviour is affected by transient faults increases with the duration of the fault.

On the other hand, faults affecting sequential elements have a greater impact in the system's behaviour than faults targeting combinational logic. Faults affecting combinational logic may not manifest as an *error* in the system because the affected combinational signal either i) traverses logic gates with other inputs in their controlling state (logical masking), or ii) is outside the latching windows of all the FFs in the path (temporal masking), or iii) is attenuated by the limited bandwidth of the technology (electrical masking) [112].

FADES and VFIT obtained very similar results when dealing with *bit-flips* and *pulses*. However, the different approaches used to inject *stuck-ats*, *indeterminations*, and *open-lines* cause their results not to be so close.

Differences related to *stuck-at* faults are mainly due to the consideration of what an injection point is. For instance, FADES considers every single FF as an eligible injection point for faults affecting the sequential logic of the system's model. However, VFIT determines these same injection points according to the signals and variables of the model. For instance, an 8-bit register is considered as a single injection point for VFIT and as 8 different injection points in FADES. As faults have been injected uniformly among all the selected injection points, this leads to 14 sequential injection points (registers) for VFIT and 81 sequential injection points (FFs) for FADES. This different distribution causes the disparity in the results.

When dealing with *indeterminations*, VFIT changes the logical value of the affected element by `'X'` (unknown value). FADES is unable to modify the voltage level of an element to somewhere in between the high- or low-level thresholds of the technology (see Section 4.6). Then, FADES randomly determines the final logic value (`'0'` or `'1'`) caused by the fault and forces the state of the affected element to that value. That different assumption, also contributes to the difference in the results.

In case of injecting *open-lines*, VFIT has opted to modify the logical value of the targeted element by `'Z'` (floating value). This methodology disconnects all the elements along the path. The approach followed by FADES consists in physically opening the line to inject the fault into the system. Depending on the selected injection point, one, several, or all the logic elements connected to the affected path may get undriven. This is the main reason why FADES presents a lower rate of affected experiments by *open-line* faults.

It is also remarkable that FADES always obtains a lower percentage of *failures* when dealing with faults affecting the system's combinational logic.

VFIT selects the possible injection points among the signals and variables of the model representing combinational logic. These points, once implemented onto an FPGA, correspond to a number of LUTs representing the structure of

the combinational circuit. For instance, `sum <= in1 + in2;`, which represents a combinational adder, could be affected by faults at *sum*, *in1*, and *in2*, when using VFIT. However, assuming that this code is implemented as a half adder, FADES can also target the *XOR* and *AND* logic gates apart from the input and output signals. In this case study, this leads to a total of 131 combinational injection points (signals and variables) for VFIT and 5283 combinational injection points (LUTs' inputs, outputs, and internal structure) for FADES.

Moreover, faults affecting this logic can be nearly only masked by logic when using VFIT. On the other hand, as FADES provides a physical implementation of a prototype, logic, temporal and electrical masking may attenuate the injected faults.

It explains the higher rate of affected experiments that VFIT achieves when dealing with faults targeting the combinational logic of the system.

FADES' results, although somewhat different than the ones provided by VFIT, follow the same trend and seem to be coherent with the common knowledge about faults and their manifestation. This establishes the correctness of the implementation of this first prototype and all the proposed approaches for injecting hardware faults by means of FPGAs.

6.3 Speeding-up experiments' execution

The main goal of the implemented methodology is to accelerate the execution of simulation-based fault injection experiments for a wide set of hardware fault models. Therefore, it is still necessary to ascertain whether this primary goal has also been achieved.

In order to estimate the attainable speed-up for each of the considered hardware faults, the execution time of the fault injection experiments presented in the previous Section 6.2 was measured. Table 6.3 lists the execution time, in seconds, of the experiments carried out by FADES and VFIT.

Results from VFIT show that the execution time of experiments injected by using the *simulator commands* technique is constant and equal to 21600 seconds. This technique simply makes use of the commands supplied by the simulator to change the logic value of the targeted signal, which has a constant temporal cost regardless the fault being injected.

However, the execution time of fault injection experiments using FADES is dependent on the fault model being considered. As presented in Chapter 4, the emulation time depends on the information that must be transferred to and from the FPGA to change its configuration memory.

Table 6.3: *Execution time (in seconds) required to inject 3000 faults by means of FADES and VFIT.*

Fault model	FADES				VFIT
	Transient faults		Permanent faults		
	Sequential logic	Combinational logic	Sequential logic	Combinational logic	
Bit-flip	916	-	-	-	21600
Pulse	-	755/1520	-	-	21600
Indetermination	1065	805	1043	625	21600
Delay	2487	2778	2076	2267	21600
Stuck-at	-	-	912	641	21600
Stuck-open	-	-	1322	1585	43200
Short	-	-	2099	2497	43200
Open-line	-	-	2032	2165	21600
Bridging	-	-	2158	2317	43200

The attainable speed-up according to the injected fault, its duration, and the logic element targeted is summarised in Table 6.4.

Table 6.4: *Speed-up ratio attained by FADES with respect to VFIT.*

Fault model	Transient faults		Permanent faults	
	Sequential logic	Combinational logic	Sequential logic	Combinational logic
Bit-flip	23.58	-	-	-
Pulse	-	28.60/14.21	-	-
Indetermination	20.28	26.83	20.70	34.56
Delay	8.68	7.77	10.40	9.52
Stuck-at	-	-	23.68	33.69
Stuck-open	-	-	32.67	27.25
Short	-	-	20.58	17.30
Open-line	-	-	10.62	9.97
Bridging	-	-	20.01	18.64

As can be seen in Table 6.4, FADES achieves a better speed-up ratio for experiments targeting the logic of the system (*bit-flip*, *pulse*, *stuck-at*, *stuck-open*, and *indetermination*) than when targeting the routing of the system (*short*, *open-line*, *bridging*, and *delay*). Logic-related faults attain a global speed-up ratio of 26.80 whereas routing-related ones achieve a global 9.45 ratio.

That great difference in the experiments' execution time is caused by the current implementation of the prototype. Logic-related faults follow the proposed methodology for fault emulation, which can be briefly described as i) reading some information from the FPGA (if any), ii) generating and downloading a new reconfiguration file for injecting the fault, and finally iii) gene-

rating and downloading another reconfiguration file to restore the fault-free configuration of the system to remove the fault.

However, we have detected a problem when modifying the routing of the system implemented on the prototyping board. Once the FPGA has been reconfigured to restore the fault-free configuration of the system, its configuration memory states that the internal connections have been correctly changed. But, somehow, the routing segments or the interconnecting pass transistors seem to keep some capacity charge and, hence, the system is not properly behaving anymore. The only way of bypassing this problem has been resetting the prototyping board to turn off and discharge the programmable device.

Hence, routing-related faults follow this other approach for injecting a fault: i) reading the *whole* current state of all the sequential elements of the system, ii) resetting the prototyping board, iii) generating and downloading a new full reconfiguration file to inject the fault, iii) restoring the previous state of all those sequential elements by using a *stuck-at*-like approach, and at last iv) resetting the board and downloading the original configuration file to remove the fault from the system. This obviously involves transferring much more information than in the case of logic-related faults. Thus, it increases the execution time and reduces the attainable speed-up.

Another technical problem with our prototyping board is the cause of not so good (one order of magnitude) speed-up ratio. After downloading a *partial* reconfiguration file onto the available FPGA, the contents of its configuration memory do not match the contents specified by the reconfiguration file. In some situations, modifying part of the logic of the programmable device causes some unrelated logic to change too, leading to an undesired behaviour.

This problem has been bypassed by including an option in FADES that allows the user to select whether to generate and use *partial* or *full* reconfiguration files. All these experiments were carried out using a *full* reconfiguration approach. This greatly increases the amount of information transferred to and from the FPGA, thus slowing down the execution of the experiments and reducing the attainable acceleration.

Nevertheless, there is plenty of room for including different optimisations that have been successfully used for simulation-based fault injection and also for fault emulation by means of Compile-Time Reconfiguration [54], such as restoring the system's state before the last injection instead of running again till that point, or ending the emulation once the fault has been positively classified. We expect to increase the attainable acceleration well beyond two orders of magnitude by implementing some of these optimisations, by following a *partial* reconfiguration approach and solving some other technical problems caused by the current implementation of this first prototype.

FADES' results encourage the idea of using FPGAs to accelerate simulation-based fault injection experiments for a wide set of representative hardware faults. Even following the worst case *full* reconfiguration approach, FADES is able to speed-up experiments' execution time by at least one order of magnitude. This shows the feasibility of the proposed methodology and the different approaches for injecting each one the considered faults.

6.4 Methodology and tool features

As the main goals of the tool has been achieved, it could be interesting to evaluate some other attributes of the implemented methodology and tool [93]. The following subsections describe different properties of the proposed methodology and its implementation, such as execution time and its scalability, spatial intrusion, portability, independence from model description, controllability, accessibility, and observability, pointing out their advantages and drawbacks.

6.4.1 Execution time and scalability

Section 6.3 has already discussed the time required to execute fault injection experiments using FADES for each one of the considered hardware fault models. All these experiments targeted the same system and, therefore, the question of whether the obtained results will be of applicability to any system under study has not been answered yet.

This section deals with this issue, assessing how the time required to emulate a fault using FADES scales depending on the complexity of the system under study.

Although complexity is in some sense an intuitive concept, there is no general definition or single accepted definition of complexity when applied to a model [113]. Parameters such as the number of signals, variables, or code lines, could be representative of the *model's complexity*, but they are not really representative of the *complexity of the modelled system*.

However, once the model has been implemented on an FPGA, it is possible to obtain very detailed information regarding the number of internal resources required for its implementation. This information can be used to easily classify the systems according to the area they take into the device.

This is the key idea that has been used to evaluate the execution time of a number of fault injection experiments carried out into three different systems. These three systems were a model of a PIC microcontroller (PIC) [114], the

already presented model of an 8051 microcontroller (8051) [108], and a custom set of 24 interconnected PICs (PICNET), which was designed just to increase the complexity of the system. Table 6.5 shows the number of resources required to implement each one of these systems.

Table 6.5: *Complexity of systems in terms of number of FPGA's internal resources required for their implementation.*

System's model	#FFs (%FFs)	#LUTs (%LUTs)	#Slices (%Slices)
PIC	335 (1.36%)	277 (1.12%)	305 (2.48%)
8051	635 (2.58%)	5310 (21.60%)	2882 (23.45%)
PICNET	8155 (33.18%)	7071 (28.77%)	7466 (60.75%)

The systems are classified according to the three different kinds of FPGA's internal resources considered: i) the number of FFs, which represents the complexity of the sequential logic of the system, ii) the number of LUTs, which estimates the complexity of the system in terms of its combinational logic, and iii) the number of *slices*, which assesses the global complexity of the system, as *slices* consists of FFs and LUTs either used for implementing logic or routing the system. As can be seen in Table 6.5, and according to the number of resource used for their implementation, the PIC is the least complex system, followed by the 8051 model and, finally, the PICNET system is the most complex one.

A fault injection campaign was carried out for each one of these systems. It consisted in 3000 experiments for each one of the considered hardware fault models affecting the sequential and combinational logic of the system. Faults were uniformly injected along the space and time.

During the execution of these experiments, the time required to perform a number of processes was measured: i) the time devoted to read some information from the FPGA required to emulate the fault (readback), ii) the time spent reconfiguring the FPGA to inject the fault (injection), iii) the time required to reconfigure the FPGA to restored the fault-free configuration of the system (deletion), iv) the time consumed while monitoring the state of the system (observation), and v) the time spent just controlling and managing the whole fault injection process (management). The total execution time of a fault injection experiment consists in the sum of all these partial times.

Figures 6.1 and 6.2 depict the mean time devoted to execute the defined processes for each of the considered systems. As logic-related faults present better execution times than routing-related ones (see Section 6.3), they have been respectively presented in Figures 6.1 and 6.2.

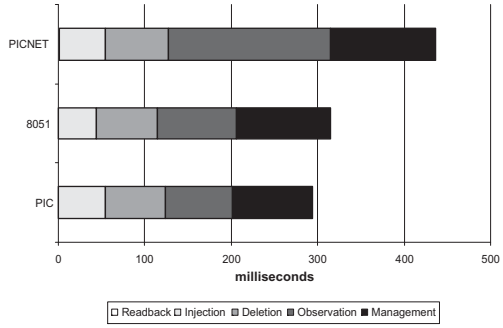

Figure 6.1: *Mean time required to emulate logic-related faults (*bit-flips*,* pulses*,* stuck-ats*,* stuck-opens*, and* indeterminations*).*

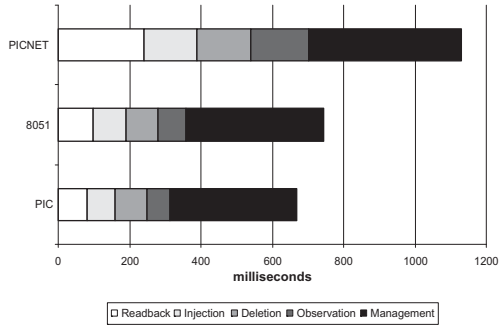

Figure 6.2: *Mean time required to emulate routing-related faults (*shorts*,* open-lines*,* bridgings*, and* delays*)*

Regarding faults affecting the logic of the system, it can be seen in Figure 6.1 that neither the readback, injection nor deletion times depend on the complexity of the system. These times obviously depend on the kinds of faults being injected but, on average, they keep nearly constant and are not really affected by the system's complexity.

On the other hand, the observation time increases with the system's complexity. The observation process obtains the current state of the system, which can be defined as the logic value of *all* the registers (FFs) of the system. According to this, it will be necessary to transfer a higher amount of information for system comprising a higher number of FFs. Therefore, the observation time scales with the number of FFs in the system.

The management time slightly increases with the complexity of the system. Eligible fault injection points are selected among the system's FFs and LUTs. More complex systems have more injection points which slightly lengthen the overall management process.

Hence, it can be concluded that the time FADES requires to perform fault injection experiments targeting the logic of the system's model, scales with the number of FFs used for its implementation.

As can be seen in Figure 6.2, all the considered partial times (readback, injection, deletion, observation, and management) for routing-related fault emulation seem to increase with the complexity of the system.

Routing-related fault injection experiments require longer executing times than logic-related ones, due to the particular problems the current implementation of FADES tries to solve (see Section 6.3).

First of all, the whole state of the system (the current state of all the FFs used in the system's implementation) must be retrieved before injecting the desired fault. So the readback time increases with the system's complexity in terms of number of FFs.

Both the injection and deletion processes must reset the prototyping board, download a full reconfiguration file and, after that, restore the previous state of the system. Restoring this state depends again on the number of registers the system consists of. Therefore, the injection and deletion times scale with number of FFs used in the system's implementation.

The observation and management times follow the same trend already explained for logic-related faults. Hence, they also depend on the number of FFs required to implement the system's model.

Then, the time FADES requires to execute fault injection experiments targeting the routing of the system's model scales with the complexity of that system in terms of the number of FFs used for its implementation.

Taking all this information into account, it can be concluded that the execution time of fault injection experiments, using FADES, mainly scales with the complexity of the system under study in terms of the number of FFs required for its implementation.

It is to note that Virtex FPGAs present coarse-grain partial reconfiguration capabilities and, hence, the minimum readback unit is not a single FF but a whole column of FFs (*frame*). Then, it is not really the number of FFs but the number of *frames* what determines the execution time of fault injection experiments in this prototype. The final allocation of sequential elements on the FPGA may cause that systems with similar number of FFs present different number of used *frames* and, thus, different experimentation time. This is an important issue that can guide the implementation of the system onto the FPGA to reduce the execution time of fault injection experiments and, therefore, increase the attainable speed-up.

On the other hand, one quarter to half of the total execution time is devoted to controlling and managing the whole process. This clearly states that the current implementation of FADES has wide room for data structures and algorithms optimisation, thus reducing even more the overall execution time.

6.4.2 Spatial intrusion

Spatial intrusion makes reference to what extent a system must be modified to allow for the injection of faults.

As the proposed methodology works with the final implementation of the system's model, it is not modified at all. However, this implementation has to be somewhat modified to be able to inject some of the considered hardware faults.

The injection of *bit-flip* faults presents no spatial intrusion into the system. Although some of the resources of the FPGA are used to inject the fault, all of them remain as in the original implementation once the current state of the sequential element has been flipped.

Faults like *pulses*, or *stuck-ats*, *stuck-opens*, and *indeterminations* into combinational logic, also present no actual spatial intrusion. The injection of these faults is based on modifying the contents of already used LUTs, and no additional resources are required.

The injection of *stuck-ats*, *stuck-opens*, and *indeterminations* into sequential logic introduces a very low spatial intrusion. Free resources, such as a LUT or LSRin line, must be used to hold the current state of the targeted FF.

All the routing-related faults, but the *open-line*, present a higher spatial intrusion than the rest of faults. Some different unused elements such as FFs, LUTs, and pass transistors, are required to modify the routing of the system to emulate its behaviour in the presence of the desired fault.

On the whole, the current implementation of the proposed methodology introduces a low spatial intrusion, if any, on the implementation of the system on the FPGA, but none in the original model. This intrusion depends on the kind of fault being considered.

6.4.3 Portability

The portability of FADES can be studied from two different perspectives related to its hardware and software components.

FADES currently supports the RC1000 board [105] from Celoxica[2] and the XSV800 board from Xess[3]. It can easily support any other prototyping board holding an FPGA from the Virtex family by developing the required XHWIF interface to access all the resources of the board by means of the JBits package.

The use of other families of FPGAs will require the development of a package analog to JBits to enable the easy access and configuration of the internal resources of the selected FPGAs family.

On the other hand, the software modules of FADES have been entirely written in the Java programming language. This ensures its portability to any platform with an available Java Virtual Machine.

Moreover, FADES can be easily extended to provide remote access to any available prototyping board. This feature, supported by the XHWIF interface, enables researchers to share not so cheap prototyping boards in an effective way.

Therefore, the portability of FADES counts as a strong point in its favour.

6.4.4 Independence from model description

Most model-based fault injection tools require the use of certain HDL languages or description levels to model the system under study. For instance, FIFA [50] uses gate-level models, whereas VFIT [31] only supports VHDL models.

The use of a Run-Time Reconfiguration approach for the injection of faults results in the proposed technique to be independent from the HDL language and description level used to model the system under study. Any language, such as VHDL, Verilog, SystemC, and even schematic or finite state machine diagrams, can be used to specify the system under study. The only actual requirement is that the system's model must be synthesisable. This allows the use of a wide variety of already existing models, mixing languages and description levels to use the best suitable for the component being modelled.

FADES can also manage IP (Intellectual Property) cores, which usually provide a relocatable placed and routed component but not its source code.

6.4.5 Controllability and accessibility

Simulation-based fault injection techniques usually present very high controllability and accessibility, since commercial simulators grant access to every single aspect of the model (signals and variables). FPGA-based fault injection tools present a somehow more restricted controllability and accessibility.

[2] http://www.celoxica.com/
[3] http://www.xess.com/

As previously explained, FADES controls the clock signal of the system, being able to stop the system's execution and resume it later. This is the easiest solution to asynchronously inject faults using inherently synchronous FPGAs. Although this approach has a very coarse grain temporal precision, just somewhere between two clock cycles, the fault is virtually injected at the precise time it was supposed to occur.

This also grants the repeatability of experiments, since the state of the system can be stored and later recovered to resume an experiment.

The accessibility is limited to those configurable elements that can be directly accessed through the FPGA's configuration memory. This is usually enough to manage most of the FPGA's internal resources, like LUTs, memories, multiplexers, and pass transistors. However, the state of elements such as FFs cannot be directly changed, and some procedures like the proposed for the injection of *bit-flips* (see Section 4.2) must be followed to achieve that goal. Some other elements of modern FPGAs, such as multipliers or microprocessor cores, also present difficulties when trying to access all their internal elements.

The injection of some faults, such as *indeterminations* are also problematic, since working with digital systems that only allow to induce a well-defined `'0'` and `'1'` in an element or line.

On the other hand, FADES achieves higher degrees of controllability and accessibility than classical prototype-based fault injection techniques, like pin-level and heavy-ion radiation.

The controllability and accessibility of FADES is, then, also satisfactory.

6.4.6 Observability

According to its synchronous nature, Virtex FPGAs can provide the current state of their synchronous elements and the configuration of the whole system. Despite FADES can only monitor the system's synchronous elements, as nearly all modern systems are synchronous, this is not really a problem.

As detailed in Section 6.4.1, the main drawback of this approach is the temporal overhead caused by the readback process. Reducing the number of observation points and/or decreasing the monitoring rate will result in a better speed-up ratio at the cost of decreasing the observability of the system.

The different trace schemes implemented by FADES can be used to adjust the monitoring rate and observation points to suit any needs.

Hence, in general, FADES achieves a high observability although it directly affects the temporal intrusion introduced in the execution of the experiments.

6.5 Case Study

Once the feasibility of the proposed approach has been shown, and the main features of the current implementation has been presented, it could be desirable to show some examples of the possible application of this methodology to support the dependability assessment of embedded digital systems [115].

The use of Components-Off-the-Shelf (COTS) in the design of embedded systems is currently a hot topic. It offers reduced time-to-market costs and rapid integration of technology innovations in embedded solutions.

The best standardised set of benchmarks for embedded systems is defined by the Embedded Microprocessor Benchmark Consortium[4] (EEMBC). It copes with a wide range of embedded applications and performance requirements, targeting automotive/industrial, consumer, networking, office automation, and telecommunication domains. Table 6.6 lists the set of benchmarks defined for each one of these application domains.

Table 6.6: *The EEMBC benchmark suite.*

Application domain	Considered benchmarks
Automotive/ Industrial	6 microbenchmarks (arithmetic operations, pointer chasing memory performance, matrix arithmetic, table lookup, and bit manipulation), 5 automobile control benchmarks, and 5 Fast Fourier Transformation (FFT) filtering benchmarks
Networking	5 multimedia benchmarks (JPEG compress/decompress, filtering, and RGB conversions)
Consumer	Shortest path, IP routing, and packet flow operations
Office automation	Graphics (Bézier curve calculation, dithering, and image rotation) and text processing benchmarks
Telecommunication	Filtering and DSP benchmarks (autocorrelation, FFT, decoder, and encoder)

This set of benchmarks can be used to determine which is the most suitable microprocessor, from a given set, to be integrated into a system in a certain application domain. This decision is only based on the performance evaluation of the selected microprocessors.

[4] http://www.eembc.org/

However, when dealing with critical embedded systems, not only the performance but also the dependability attributes of those microprocessors should be estimated. In this context, FADES could be used to assess the dependability of these microprocessors and allow for their comparison in terms of the obtained results.

This Section presents the results obtained by FADES when assessing the dependability of three different processing cores for a particular application domain. These results are interpreted to determine which is the most suitable core depending on several different factors, such as the kind of system under study, the probability of occurrence of faults, or the ratio between the combinational and sequential logic of the considered cores.

6.5.1 Cores under study

The dependability of three processing cores, featuring a very different architecture, has been analysed to determine the suitability of their integration into an embedded system to perform the Gaussian smoothing operation.

The model of the PIC microcontroller [114] presents a RISC (Reduced Instruction Set Computer) architecture that takes between 1 and 4 clock cycles to process each instruction. It embeds an integer Arithmetic-Logic Unit (ALU) and an 8-bit width data path.

The previously used model of the 8051 microcontroller [108] has also been analysed. It features a CISC (Complex Instruction Set Computer) architecture that can execute branch instructions in 8 clock cycles and the rest of instructions in only 4 clock cycles. As the PIC microcontroller, it also embeds an integer ALU with and 8-bit data path.

The third core is a custom implementation of a specific-purpose DSP-like architecture. It is able to process up to three MAC (Multiply-and-ACcumulate) fixed-points operations in parallel, and presents a 16-bit wide data path.

The integration of these cores into a system was performed by using dual-port memories, a very common approach to most embedded systems. These memories act as interfaces were the microprocessors and other components of the system can store and read data for their communication. Hence, the image acquired by the system is stored into a dual-port memory where the core under study can retrieve it. The output of the image processing algorithm will be stored into another dual-port memory allowing other components of the system to access this information.

6.5.2 Workload

As can be seen in Table 6.6, signal filtering operations are considered for benchmarking in four out of the five domains defined by the EEMBC benchmark suite. Typically, these operations include a 2D convolution operator.

If an image I consists in M rows and N columns of pixels, and a kernel K consists in m rows and n columns of coefficients, then a 2D convolution operator may be described as shown in Equation 6.1, where i runs from *1* to *M-m+1* and j runs from *1* to *N-n+1*.

$$O(i,j) = \sum_{k=1}^{M} \sum_{l=1}^{N} I(i+k-1, j+1-1) K(k,l) \tag{6.1}$$

A 2D convolution operator named *Gaussian smoothing*, which is commonly used to reduce the noise of incoming signals, has been selected as a representative workload for embedded systems. This operator applies a 3x3 pixels kernel K (see Equation 6.2) to an incoming image I to remove detail and noise.

$$K = \frac{1}{16} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{6.2}$$

An example of the application of this operator to an incoming image is shown in Equation 6.3.

$$Gaussian \begin{pmatrix} 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \\ 5 & 10 & 15 & 20 \\ 7 & 14 & 21 & 28 \end{pmatrix} = \begin{pmatrix} 6.5 & 9.75 \\ 10 & 15 \end{pmatrix} \tag{6.3}$$

6.5.3 Faultload

As faults were uniformly distributed across space, the number N of experiments targeting the logic of the system for each of the considered hardware faults was computed according to Equation 6.4. In that expression P represents the probability of activating one of the injection points with the lowest probability of activation. Hence, Equation 6.4 assures that each possible injection point will be statistically targeted at least once with a probability Q.

$$N = \frac{ln(1-Q)}{ln(1-P)} \tag{6.4}$$

Taking into account that *bit-flips* and *pulses* can only target sequential and combinational elements, respectively, and that routing-related faults may affect any kind of logic, there are 10 different kind of faults that can target each kind of logic.

According to this, and assuming that all the eligible injection points can be targeted with the same probability, the number of experiments performed to ensure that each injection point will be targeted at least once with a probability of 0.99 is shown in Table 6.7.

For instance, the PIC requires 721 FFs and 2116 LUTs to be implemented onto the Virtex FPGA, so the probability of the fault targeting one of these elements is 0.00138 ($P = \frac{1}{721}$) and 0.0004725 ($P = \frac{1}{2116}$), respectively. So, after applying Equation 6.4, it leads to 3319 experiments for each kind of fault targeting the sequential logic of the system and 9743 for faults targeting the combinational logic of the system. In the end, a total of 130620 experiments were performed on this system.

Table 6.7: *Number of experiments performed depending on the system's complexity.*

Core's model	#FFs	#LUTs	#considered HW faults	#experiments per HW fault		Total #experiments
				Sequential	Combinational	
PIC	721	2116	10	3319	9743	130620
8051	641	4208	10	2950	19377	223270
DSP	1105	2649	10	5087	12197	172840

All these faults were also injected following a uniform distribution along the experiment duration.

6.5.4 Measures

All the sequential elements of the system's model (FFs and memories) were monitored at the end of each fault injection experiment to obtain the *state* of the system at that particular point in time.

From this whole set of observation points, those related to the *output* of the Gaussian smoothing operation (stored in a dual-port memory) where considered as the most significant ones for determining the behaviour of the system in the presence of faults.

Different works already defined convenient failure modes classifications, such as the CRASH scale [116] (*C*atastrophic, *R*estart, *A*bort, *S*ilent, and *H*indering failures) or the MAFALDA [24] classification (according to three distinct observation levels). Nevertheless, neither of them can be easily applied in the context of the proposed case study.

On the one hand, the selected workload (Gaussian filtering) is not supported by a proper operating system, but is directly executed by the considered microprocessor. On the other, the models of the cores under study do not implement any kind of error detection mechanisms.

Hence, according to the measures provided by FADES (see Section 5.5), results can be divided into the following custom failure modes, which are generic enough to be applied to any system under analysis.

- **Failure.** The output of the faulty system is not equal to that of the Golden Run and, hence, the system is not providing the expected service to the user. This is assimilated to those experiments with a severe impact on the behaviour of the system.
- **Latent.** In this case, although the system is providing its expected service correctly, the internal state of the system is not correct. This can lead to a *failure* in an undetermined amount of time. It can be assimilated to experiments with a moderate impact on the system's behaviour.
- **Masked.** The outputs and the state of the system are correct and so, the system has tolerated the fault. It is obviously related to experiments in which faults have no effect, probably due to being somehow masked.

The percentage of experiments falling into each of these three failure modes will be reported to the user, for each type of fault, according to the kind of logic targeted by the faults and the nature of the faults.

6.5.5 Analysis of results

The results of all these experiments were analysed by FADES to determine the percentage of experiments leading to the different defined failure modes for each considered hardware fault.

Figure 6.3 depicts this information for the PIC, 8051 and DSP cores. It is to note that, although transient faults were injected with a duration of less than one clock cycle, and ranging from one and ten clock cycles, and from ten and twenty clock cycles, the results reported in Figure 6.3 are carried out considering the whole set of performed experiments for each particular transient fault.

These results could be interpreted in many different ways depending on the purpose of the designed system.

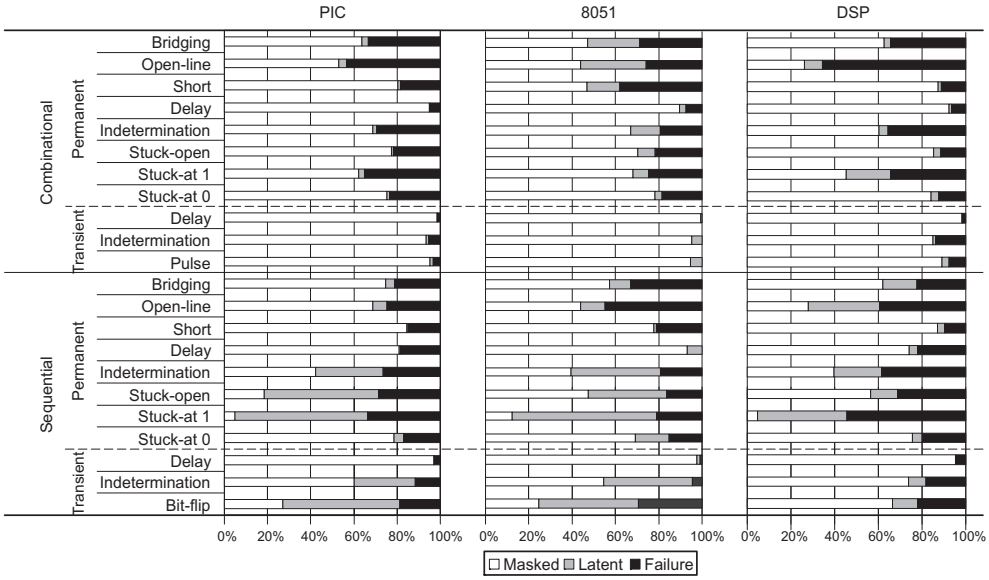

Figure 6.3: *Syndrome analysis of each core when executing a Gaussian smoothing algorithm.*

For instance, mission critical systems usually operate in hostile environments. There, *availability* may be defined as the fraction of time that the system spends performing useful work, where useful work is time spent performing computation on the application that will never be redone due to a failure. So, availability is the major concern for this type of systems. According to the previously defined failure modes, experiments leading to a *failure* have a negative impact on the availability, but *latent* and *masked* experiments provide the correct service and thus the system's availability is not affected.

On the other hand, safety critical systems are computer-based systems in which the occurrence of faults may endanger human lives or the environment. For these systems *safety* is a dependability attribute of prime importance. *Latent* experiments, which may cause a *failure* after some time, and *failure* experiments may hypothetically lead to catastrophic consequences. Although it is not possible to determine, from the information available, the effect of those failure modes on the system's environment, it is clear that they put the system into *unsafe* states that must be entirely avoided. *Masked* experiments do not affect the safety of the system, since it has tolerated the fault.

These two different points of view have been adopted to study the results provided by FADES (they must not be taken for *availability* and *safety* as defined in [2]). This study will assist the user in determining the best suitable core to be integrated into a system depending on its particular purpose.

The first thing that can be seen from Figure 6.3 is that the study of each particular case (a certain permanent/transient fault affecting the sequential/combinational logic of the system) provides so much information that it could be very difficult to manage. Unless a very specific case has to be considered it seems more natural to group the results according to the faults duration and/or the logic affected by the faults. This limits the variables the user has to deal with to obtain meaningful, concise and precise information to evaluate and compare the cores under study.

The analysis of the impact of faults into the system is next presented, beginning with particular cases and going to more general ones at the end.

6.5.5.1 Impact of transient faults on the sequential logic of the system

As shown in Figure 6.3 *bit-flips* are the transient faults with a deeper impact into the sequential logic of the system. Hence, they have been considered as the most representative faults falling into this category, and their effects on the system's behaviour are detailed in Figure 6.4.

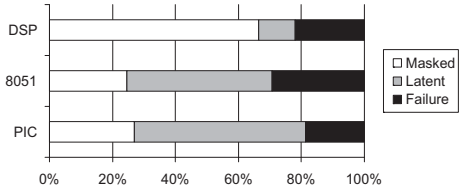

Figure 6.4: *Impact of* bit-flips *onto the system's behaviour when executing a Gaussian smoothing algorithm.*

The analysis of this kind of faults is very interesting specially in space applications where they appear due to cosmic radiation (high energy neutrons and protons) causing Single Event Upsets (SEUs) in memory cells.

In case of manned missions, safety is a major concern and, thus, *masked* experiments are related to safe behaviours. Figure 6.4 shows that the DSP core is the safest by far (66.5%), while the PIC (26.9%) and 8051 (24.7%), in this order, fall way behind.

In unmanned missions, like space probes or satellites, the availability of the system should be maximised. So, the system featuring the lowest *failure* rate (the highest *masked* plus *latent* behaviour) could be considered as the best choice. The most suitable core is the PIC (81.3%), followed by the DSP (78.0%) and 8051 (70.6%).

In the case of the PIC the large percentage of *latent* experiments (54.3%) compensates the low rate of *masked* ones (26.9%). This is related to the general purpose architecture of the microcontroller and the selected workload. A large number of registers, mainly from the bank register and some output ports, accounting for the 51% of the FFs of the microcontroller are not used during the workload execution. Hence, *bit-flips* targeting these registers do not really affect the outcome of the computation although the induced logic value will remain there until the end of the experiment (assuming a system's reset). The 8051 presents a similar behaviour.

However, the DSP-like core has been customised to perform up to 3 MAC operations in parallel and the workload really fits this specific architecture. Most of the registers are updated every single clock cycle and, hence, the low rate of *latent* experiments (11.5%). A *bit-flip* targeting these registers is very likely to be *masked*, because the induced value is rewritten due to the dynamics of the system before being really used, or result in a *failure* due to data corruption with lower probability (22.0%).

6.5.5.2 Impact of permanent faults on the sequential logic of the system

According to Figure 6.3, no permanent fault affecting the sequential logic of the system has a greater impact on the system's behaviour than other. Hence, the average impact of all these faults has been depicted in Figure 6.5 to allow for its easier analysis.

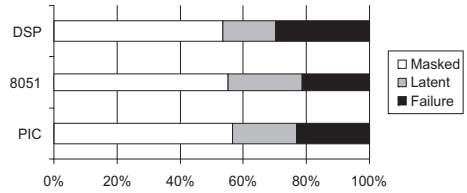

Figure 6.5: *Impact of permanent faults affecting the sequential logic of the system while executing a Gaussian smoothing algorithm.*

Permanent faults can be related to manufacturing defects and wearout mechanisms during system's normal operation. Electromigration, over-voltage, and over-current are typical mechanisms found in industrial environments with a high noise level. The steady exposition to high energy radiation may also cause permanent faults in space applications.

Under these assumptions, the 8051 core presents the lowest *failure* rate (21.3%), with the PIC close behind (23.1%), and it should be considered as the best option when improving the availability of the system is a must.

The analysed cores present close rates of *masked* experiments (PIC - 56.7%, 8051 - 55.1%, DSP - 53.4%). The PIC, which is slightly ahead, minimises the probability of reaching an unsafe state.

It is to note the difference in the system's behaviour that can be observed when considering the occurrence of *stuck-at-0s* and *stuck-at-1s*. This is, once again, mainly due to the considered workload and the cores architecture.

A large number of register's bits hold a low logic level along the workload (PIC - 78%, 8051 - 69%, DSP - 70%), so the high rate of *masked stuck-at-0* faults. However, the occurrence of *stuck-at-1s* is more harmful, especially for the DSP, since most of the registers are used to compute the workload. In the case of the PIC and 8051 this is reflected by a higher rate of *latent* experiments. It is also remarkable the relatively high rate of *masked stuck-at-1s* for the 8051, since the 3% of the register hold a high logic level for the duration of the workload.

Thus, the great importance of properly choosing a representative workload for the considered application domain.

6.5.5.3 Impact of transient faults on the combinational logic of the system

Transient faults targeting combinational logic have a very low impact on the system's behaviour (see Figure 6.3). This effect is related to three different intrinsic masking mechanisms [112] previously explained in Section 6.2.2.

Pulses seem to be of increasing importance in deep-submicron manufactured systems [117]. Hence, this study focuses on this kind of faults and its results are shown in Figure 6.6.

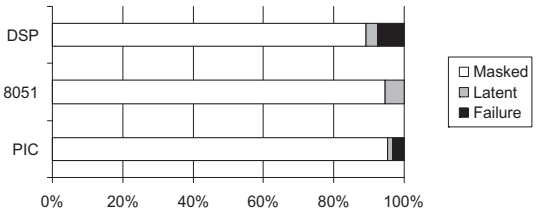

Figure 6.6: *Impact of* pulses *onto the system's behaviour when executing a Gaussian smoothing algorithm.*

Pulses, which model transient modifications of the logic level of combinational elements (SET - Single Event Transients), may occur due to radiation in space applications or electromagnetic noise (crosstalk and power spikes) in industrial environments.

The 8051 core barely presents any *failure* experiments, offering the best results from an availability perspective.

Regarding the system's safety, both the 8051 and PIC cores achieved very similar *masked* rates (95.3% and 94.6%) and therefore they could be indistinctly selected.

It is remarkable that the DSP exhibits the lowest *masked* rate (89%). This could be related to the architecture of these cores. Most of the DSP registers are updated every single clock cycle, whereas registers in the PIC and 8051 only refresh their contents when allowed by the control unit. That increases the chances of masking the fault due to temporal masking.

6.5.5.4 Impact of permanent faults on the combinational logic of the system

Figure 6.3 shows that different permanent faults affecting combinational logic, such as *stuck-at*, *indetermination*, *short*, and *open-line*, could be considered as the most dangerous one depending on the considered system and the attribute (availability/safety) to be improved. This dispersion leads us to obtain a single value, depicted in Figure 6.7, consisting in the average result for all these faults.

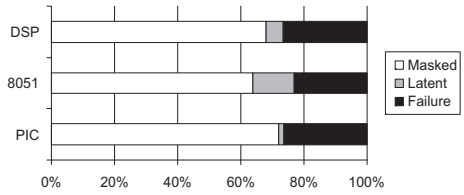

Figure 6.7: *Impact of permanent faults affecting the combinational logic of the system while executing a Gaussian smoothing algorithm.*

The 8051 presents the lowest *failure* rate (23%) and thus is the best candidate for its integration in a high-availability system. The DSP and PIC cores, with similar results (26.3% and 26.5%), are a second option.

However, when dealing with the system's safety, the PIC core presents the lowest probability of reaching an unsafe state (71.9%). In this case, the DSP core ranks in second place (67.9%).

It must be noted that, since dealing with permanent faults, the DSP is not handicapped by its lower probability of temporally masking the faults.

6.5.5.5 Impact of faults targeting the combinational logic of the system versus faults targeting its sequential logic

As shown in Figure 6.8, and as can be expected, faults affecting the sequential logic of the system have a deeper impact in the behaviour of the system than faults affecting its combinational logic.

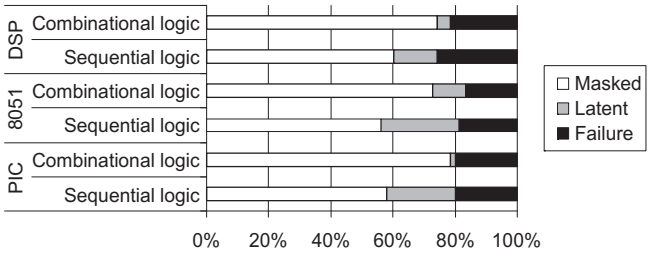

Figure 6.8: *Impact of faults affecting the combinational and sequential logic of the system while executing a Gaussian smoothing algorithm.*

On one hand, faults targeting combinational logic may be masked by any of the three masking mechanism previously presented [112] thus reducing their failure rate. On the other hand, registers may memorise *errors* that could manifest later as *failures* which obviously increases their *latent* or *failure* rate.

When considering the occurrence of faults into sequential logic, the 8051 and PIC exhibit the best behaviour from an availability perspective (81.4% and 80.2%). However, the DSP core obtains the lowest *failure* rate (39.7%) and is the best option when focusing on the safety of the system.

When dealing with faults targeting the combinational logic of the system, the PIC core presents the highest *masked* rate (80%) and the 8051 the lowest *failure* rate (16.7%). Therefore, they are best option for respectively improving the system's safety and availability.

It is to note that when considering a uniform spatial distribution of faults, their effect on the system's behaviour will depend on the ratio between its combinational and sequential logic. According to Table 6.7, this ratio is 2.93 for the PIC core, 6.56 for the 8051 core, and 2.39 for the DSP core.

Hence, although faults affecting the sequential logic of the system have a greater impact on its behaviour, there is a higher probability of targeting a combinational element and it should be taken into account when analysing the dependability of the systems under study.

6.5.5.6 Impact of transient versus permanent faults

As Figure 6.9 shows, permanent faults have a greater impact into the behaviour of the affected system than transient ones.

It seems reasonable that permanent faults, which do not disappear from the system, present the highest *failure* rate. On the other hand, transient faults achieve a higher rate of *latent* experiments due to the contribution of *bit-flips*, which invert the current state of a sequential element.

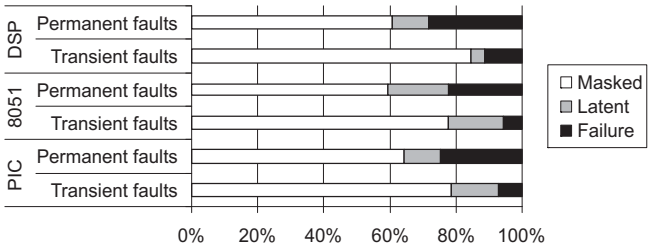

Figure 6.9: *Impact of transient and permanent faults into the system's behaviour while executing a Gaussian smoothing algorithm.*

The 8051 core presents the best *failure* rate when dealing with transient faults and, hence, it should be designated as the best candidate for improving the availability of the system. This is mainly due to its superior behaviour when affected by transient faults in its combinational logic. On the other hand, the lowest probability of reaching an unsafe state is attained by the DSP core, followed by the 8051 core.

When considering the occurrence of permanent faults, the PIC core is slightly ahead of the DSP core in terms of *masked*, but slightly behind the 8051 in terms of *failure* rates. Hence, the PIC core is the best option for improving the safety of the system and the 8051 for improving its availability.

It must be remarked that these experiments have followed a uniform distribution, but in normal component operation transient faults are more frequent than permanent ones. Likewise, permanent faults are more frequent, as shown by the bath-tube curve [118], during the initial phase (manufacturing defects) and the last phase (components wearout) of the component life cycle.

Thus, although permanent faults have a greater impact into the system's behaviour, transient ones are more frequent during its normal operation. This should also be considered to determine the effect of any fault on the system.

6.5.5.7 Summary

This section has presented an example of the applicability of the proposed methodology for assessing the dependability of a number of microprocessors cores. Measures related to *availability* and *safety* attributes have been compared to determine the best suitable core to be integrated into a system to execute a Gaussian smoothing operation.

A summary of the best core according to the duration of the injected fault, the targeted logic and the dependability attribute estimated through experimentation is listed in Table 6.8.

Table 6.8: *Best core when executing a Gaussian smoothing algorithm.*

Fault duration	Targeted logic	Best core in terms of	
		Availability	**Safety**
Transient	Sequential	PIC	DSP
Permanent	Sequential	8051	PIC
Transient	Combinational	8051	PIC
Permanent	Combinational	8051	PIC
Any	Sequential	8051	DSP
Any	Combinational	8051	PIC
Transient	Any	8051	DSP
Permanent	Any	8051	PIC

Table 6.8 shows that, in general, the 8051 core exhibits the best behaviour in the presence of faults when considering the correctness of the service provided.

On the other hand, the award as the best core to minimise the probability of reaching an unsafe state is shared between the PIC and the DSP cores. The optimal results of the DSP core when dealing with transient faults affecting the sequential logic of the system compensates the not so good results in the rest of cases and may rise this core to the first position in this particular ranking.

It is necessary to remark that these results have been obtained while executing a Gaussian smoothing algorithm. The use of a different workload could, and effectively does, change the outcome of these experiments.

For instance, these same experiments have been carried out while executing a bubblesort algorithm as workload. The DSP core was not considered, as its specific-purpose architecture can only handle MAC operations.

The results from these experiments are depicted in Figure 6.10.

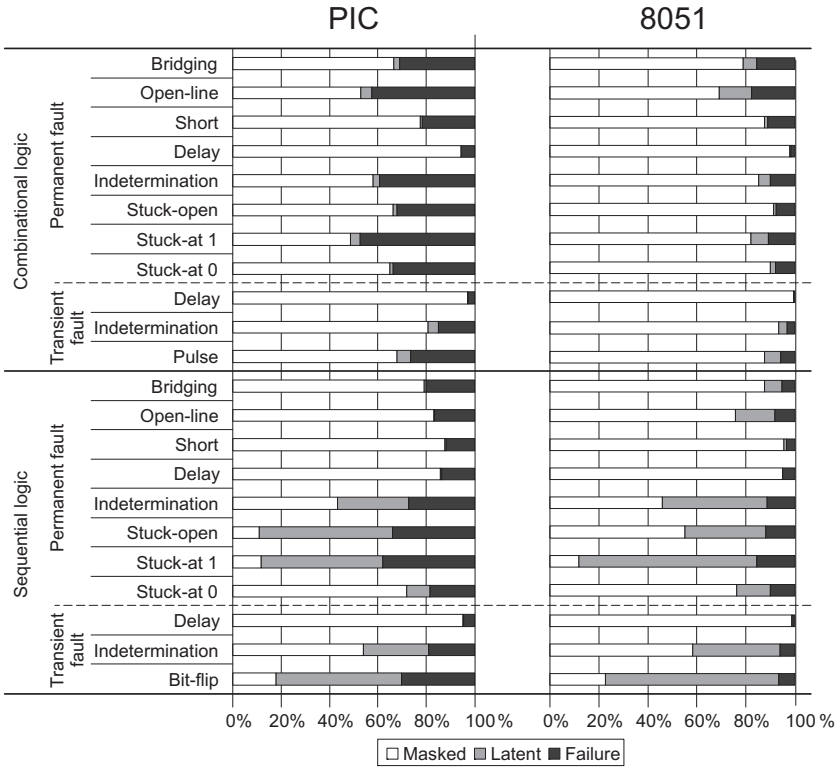

Figure 6.10: *Syndrome analysis when executing a* bubblesort *algorithm.*

As can be seen, the 8051 core surpasses the PIC core in every single case, either in terms of availability or safety. This is very interesting, as the PIC core was the best core, in terms of *masked* rate, when executing the Gaussian smoothing algorithm but cannot compete with the 8051 core when sorting integer numbers.

Hence, it is very important to correctly determine the workload to be executed while assessing the dependability of the systems under study. The selection of a workload that does not correctly represent the normal operation of the system in its working environment may lead to erroneous results and conclusions.

6.6 Conclusions

FADES is the first prototype of a fault injection tool that implements the proposed methodology for emulating a wide range of hardware faults by means of the Run-Time Reconfiguration capabilities of FPGAs.

Although the main goal of this tool is accelerate simulation-based fault injection experiments, it is necessary to keep in mind the correctness of the results provided by the tool. Therefore, FADES and VFIT, a state-of-the-art VHDL-based fault injection tool, executed the same set of experiments to allow for the comparison of their results.

Most of the results provided by both tools were very close. The main differences were related to the different characterisation of what a fault injection point is, and the different approach followed to inject faults such as *indeterminations* or *open-lines* due to technical reasons.

Then, assuming that the results obtained by VFIT are correct, these experiments validate the correctness of the proposed approaches for the injection of hardware faults and their current implementation.

Once FADES has been proved to provide confident results, it is time to determine whether it achieves the aim of speeding-up simulation-based fault injection experiments. For that reason, the time required to execute fault injection experiments by means of FADES and VFIT was also measured.

Results reflected that, on the whole, FADES achieves one order of magnitude of acceleration with respect to VFIT. Due to the different implementation of each kind of hardware fault, this speed-up ratio varies from 34.56 to 7.77. It is to note that technical difficulties, such as problems with system's rerouting or partial reconfiguration, limited the theoretically attainable speed-up.

In spite of these technical issues, results show the feasibility of the proposed approach for accelerating the execution of simulation-based fault injection experiments. There is still plenty of room for including different optimisations that have been successfully used for simulation-based fault injection and also for fault emulation by means of Compile-Time Reconfiguration. We expect to increase the attainable acceleration well beyond two orders of magnitude by implementing some of these optimisations, by following a *partial* reconfiguration approach and solving some other technical problems caused by the current implementation of this first prototype.

A number of different features related to the proposed methodology and its current implementation have also been discussed. Among them, it is to note that the experiments execution time scales with the complexity of the system in terms of the number of FFs required for its implementation onto the FPGA, it presents a very low spatial intrusion, it is highly portable, it can be applied to any system's model regardless the language and description level used in its specification, and it provides a good accessibility, controllability, and observability.

As an example of the applicability of FADES, the dependability attributes of three different cores have been assessed to determine which is the best one to be integrated into a system to execute a common signal filtering algorithm.

These cores were compared in terms of the highest probability of providing the correct service, which can be assimilated to *availability*, and the lowest probability of reaching an unsafe state, which can be assimilated to *safety*.

This study exposed a number of factors that should be taken into account when assessing the dependability of any system.

Faults appear with different probability depending on the working environment of the system under study. This should be considered to assign a different weight to the results obtained when injecting a particular kind of fault, or to avoid injecting some specific faults.

The impact of faults into the behaviour of the system greatly depends on whether combinational or sequential logic is affected by the fault. Hence, the probability of targeting a particular kind of logic, represented by the ratio between the combinational and sequential logic of the system, should also be taken into account to determine the overall effect of any fault occurring on the system.

Transient faults are more frequent during the normal operation of the system, whereas permanent ones occur more frequently during the first and last stages of the system's life cycle. This fact could be borne in mind to determine the kind of faults to be injected or to assign a different weight to each of these faults depending on the considered system's life cycle.

The selection of the workload is also a critical factor, because the results obtained will be totally dependent on the workload executed. Hence, the workload should be as representative as possible of the normal load of the system in the absence of faults.

Finally, the analysis of very specific cases, although interesting in some particular situations, may be of little interest to a common user. It should be advisable to group those results according to different criteria, such as the faults duration or the kind of logic targeted, to provide the user with concise and meaningful information that could be easily interpreted.

Chapter 7

Conclusions

There exists a great interest in developing new and efficient techniques and tools to assess the dependability of deep-submicron manufactured systems as advances in semiconductor technologies are greatly increasing the likelihood of occurrence of hardware faults.

This work has focused on the use of the Run-Time Reconfiguration capabilities of FPGAs for the early and fast dependability assessment of computer-based systems by means of fault injection.

On one hand, as following a model-based fault injection methodology, it can be applied on the first stages of the development cycle, thus reducing the costs of fixing any error in the design. It outperforms simulation-based fault injection techniques as the model of the system is run on a prototype platform rather than being executed by a software simulator.

On the other hand, other FPGA-based fault injection methodologies, like Compile-Time Reconfiguration, may achieve a better speed-up on the execution of fault injection experiments. However, they require the model of the system to be instrumented, leading to several and usually very time consuming implementations of the instrumented model onto the selected FPGA. Although the proposed methodology introduces some temporal overhead in the experiments execution, it does not require to instrument the model and, therefore, it must only be implemented once. Thus, Run-Time Reconfiguration appears as a valuable methodology for dealing with large and complex models.

Not only the probability of occurrence of hardware faults is increasing with new technologies, but also new and more complex fault models have to be considered. Currently, the evaluation of the dependability of critical system in the presence of *bit-flip*, *pulse*, *indetermination*, *delay*, *stuck-at*, *stuck-open*, *short*, *open-line*, and *bridging* faults is a must.

Nevertheless, current fault emulation methodologies mainly deal with the injection of the classical *bit-flip* and *stuck-at* fault models.

The primary goal of this thesis has been attained by defining a number of novel approaches for reconfiguring a generic FPGA architecture to emulate the behaviour of a system in the presence of a wide range of hardware faults considered representative of new deep-submicron technologies.

Several options have been considered for each hardware fault, taking into account the number of resources required to inject the fault, the time devoted to the FPGA's reconfiguration, and the applicability of such an approach.

The proposed methodology for the emulation of each fault has been detailed by using a C-like pseudo-code in Tables 4.1 (*bit-flip*), 4.2 (*stuck-at*), 4.3 (*pulse*), 4.4 (*stuck-open*), 4.5 (*indetermination*), 4.7 (*delay*), 4.8 (*short*), 4.9 (*open-line*), and 4.10 (*bridging*).

Table 4.11 summarises the different approaches that have been defined as well as the function computing the temporal cost of following each one.

The secondary goal of this thesis has been also fulfilled by developing a tool that implements the considered methodology and all the proposed approaches for injecting the whole set of hardware fault models by means of FPGAs.

FADES (***F****PGA-based Framework for the* ***A****ssessment of the* ***D****ependability of* ***E****mbedded* ***S****ystems*), which is the first prototype of that tool, has been validated by means of experimentation.

First of all, the results provided by FADES were compared with the ones provided by a state-of-the-art simulation-based fault injection tool. Results obtained from both tools were very similar, having in mind the different approaches followed to inject some faults. This establishes the correctness of the fault injection methodology and results analysis. Hence, the defined methodology can be confidently used to assess the dependability of computer-based systems.

Even in the presence of some technical problems that restricted the attainable acceleration, FADES obtained a speed-up ratio of one order of magnitude with respect to the experimentation time required by the simulation-based fault injection tool. This shows that the proposed approaches can be effectively used to accelerate simulation-based fault injection experiments and, after working out the technical issues of the current implementation, and including some optimising that have been successfully used in other methodologies, it could be possible to improve the achievable speed-up ratio well beyond two orders of magnitude.

Among its remarkable features it is to note the good portability, controllability, accessibility, and observability it provides.

One possible perspective of future research is the dependability benchmarking of VLSI systems.

The benchmarking process can be understood as the continuous measurement of a system to obtain some meaningful measures that can be used for its comparison with other systems. Traditionally the benchmarking of VLSI systems has been focused on their performance in terms of number of operations per second or another similar measure. Currently, there exists a great interest in the development of systems with very low power consumption due to very restrictive requirements in embedded systems. The increasing occurrence of faults into deep-submicron manufactured systems also claims for the development of benchmarks for the dependability assessment of critical systems.

This work has presented a study that experimentally estimates the probability of providing the correct service and reaching a safe state of different microprocessor-based embedded systems to allow for their comparison in these terms. In this sense, this constitutes a first step toward VLSI dependability benchmarking. However, an estimation of other dependability attributes such as availability, safety, reliability, integrity, and maintainability, as defined in [2], still need further research.

From all these facts arises the idea of the benchmarking of VLSI systems encompassing performance, power consumption and dependability properties. The measurement of these properties could be of great interest to determine interdependencies and finding tradeoffs among the desired level of dependability, required power consumption and minimum acceptable performance. This comparison will serve designers to improve their implementations, system integrators to select the most suitable component for a particular system, and final users to assess the capabilities of their systems.

Another perspective of the methodology is the integration of simulation-based and FPGA-based fault injection techniques into a single *co-validation* framework.

This research should determine when to use *fault simulation* or *fault emulation* depending on the characteristics of the available model, such as the language used to model the system and at which level it has been described, the type and nature of the faults to be considered, the selected locations to be affected by the faults, or the desired level of controllability, accessibility, and observability.

A step further will consist in considering the simultaneous application of both techniques to a single model. This study will encompass the model partitioning and definition of the interfaces between each part, the communication between the simulator and the FPGA, the synchronisation of the model's

execution during the fault injection process, the system monitoring and the correlation of the obtained measurements, among many others.

All these questions, once answered, may lead to the definition of very interesting design for validation rules and guidelines. This issue could nicely complement the benchmarking of VLSI systems as they will have been designed to be effectively validated in the presence of faults in an optimal way.

Finally, another possibility consists in considering the use of the Run-Time Reconfiguration methodology for the dependability assessment of FPGA-based systems. An increasing number of systems are being implemented by means of FPGAs, mainly due to the reconfigurable capabilities they offer, and the large number of IP cores that are available from third parties to develop new systems by integration of these cores.

In that case, FPGAs are not just prototyping tools to implement the model of the system and emulate the occurrence of faults, but are also the final target on which the final system will be implemented.

The analysis and definition of FPGA-related fault models will be required as a first step to the development of new approaches for emulating the occurrence of these faults into the programmable device. Surely, the knowledge we have already obtained in defining approaches for emulating different fault models will be very valuable for this research.

Appendix A

Detailed description of files and commands used by FADES

A.1 Bitstream file

The *bitstream* file consists in a series of 32-bit word command and data packets [97]. Command packets target the Virtex FPGA's control logic for managing the reconfiguration process, whereas data packets are destined to modify the FPGA's reconfigurable logic.

Table A.1 details the format of a sample configuration file for a full FPGA reconfiguration.

First of all, a header is used to synchronise the transmission between the host and the configuration logic of the FPGA. This header also contains different parameters to manage the reconfiguration process, such as the position in the configuration memory of the FPGA where the transferred information must be placed into.

For instance, as this is a full reconfiguration file, the configuration data must be placed beginning at the position related to the first CB (located in row 0 and column 0 of the FPGA). A total of 142902 data words will be transferred to fully reconfigure all the logic elements of the device.

After that, the contents of the internal memory blocks of the FPGA are also programmed. Two blocks of 2210 and 2176 data words, respectively, are downloaded into these memory blocks.

Finally, a Cyclic Redundancy Check is performed to ensure the information has been properly received, and the start-up sequence of the FPGA is initiated.

Table A.1: *Analysis of a* bitstream *file (.bit) format.*

Bitstream data	Section	Description
FFFFFFFF		Dummy Word
FFFFFFFF		Dummy Word
AA995566		Synchronisation word
30008001		Packet Header: Write to CMD (CoMmanD) register
00000007		Packet Data: RCRC (Reset CRC register)
30016001		Packet Header: Write to FLR (Frame Length Register)
00000021		Packet Data: Frame Length (33)
30012001		Packet Header: Write to COR (Configuration Option Register)
00803F2D	HEADER AND	Packet Data: Configuration options
3000C001	CONFIGURATION	Packet Header: Write to MASK (MASK register)
000001C6	OPTIONS	Packet Data: CTL (ConTroL Register) mask
3000A001		Packet Header: Write to CTL (ConTroL Register)
00000040		Packet Data: Security and Port persistence
30008001		Packet Header: Write to CMD (CoMmanD) register
00000009		Packet Data: SWITCH (change CCLK (Configuration CLocK) frequency)
30008001		Packet Header: Write to CMD (CoMmanD) register
00000001		Packet Data: WCFG (Write ConFiGuration data)
30002001		Packet Header: Write to FAR (Frame Address Register)
00000000		Packet Data: Frame Address (CLB, 0, 0)
30004000		Packet Header: Write to FDRI (Frame Data Register Input)
50022E36		Packet Header Type 2: Data words (142902 words)
00123018		Packet Data: 1^{st} Configuration word
....		Packet Data: n^{th} Configuration word
00000000		Packet Data: 142902^{nd} Configuration word
30002001		Packet Header: Write to FAR (Frame Address Register)
02000000		Packet Data: Next frame address (BRAM, 0, 0)
30004000		Packet Header: Write to FDRI (Frame Data Register Input)
500008A2		Packet Header Type 2: Data words (2210 words)
00000000		Packet Data: 1^{st} Configuration word
....		Packet Data: n^{th} Configuration word
00000000		Packet Data: 2210^{th}Configuration word
30002001	DATA FRAMES	Packet Header: Write to FAR (Frame Address Register)
02020000	AND CRC	Packet Data: Next frame address (BRAM, 1, 0)
30004000		Packet Header: Write to (Frame Data Register Input)
50000880		Packet Header Type 2: Data words (2176 words)
00000000		Packet Data: 1^{st} Configuration word
....		Packet Data: n^{th} Configuration word
00000000		Packet Data: 2176^{th} Configuration word
30008001		Packet Header: Write to CMD (CoMmanD) register
00000007		Packet Data: RCRC (Reset CRC register)
30008001		Packet Header: Write to CMD (CoMmanD) register
00000003		Packet Data: LFRM (Last FRaMe write)
30004022		Packet Header: Write to FDRI (Frame Data Register Input) (34)
00000000		Packet Data: 1^{st} Configuration word
....		Packet Data: n^{th} Configuration word
00000000		Packet Data: 34^{th} Configuration word
30008001		Packet Header: Write to CMD (CoMmanD) register
00000005		Packet START (begin START-up sequence)
30008001		Packet Header: Write to CMD (CoMmanD) register
00000007	FINAL CRC	Packet Data: RCRC (Reset CRC register)
00000000	AND START-UP	Dummy word
00000000		Dummy word
00000000		Dummy word
00000000		Dummy word

A.2 Logic Allocation file

The Logic Allocation file shows, in a proprietary format, the location of all the sequential elements (FFs and memory blocks) that have been used to implement a particular system's model on a Virtex FPGA.

Table A.2 shows a sample Logic Allocation file for a Virtex FPGA.

The first part of the file consists in a series of comments (lines beginning with ';') that explain the format used to specify the location of the sequential elements on the FPGA.

Next, lines beginning with the keyword 'Info' show information related to the options used to generate the bistream file this Logic Allocation file is related to.

The rest of the lines, beginning with the keyword 'Bit', locate each one of the stated sequential elements.

Table A.2: *Sample Logic Allocation File (.ll).*

```
Revision 3
; Created by bitgen G.31a at Wed Sep 28 12:36:02 2005
; Bit lines have the following form:
; <offset> <frame address> <frame offset> <information>
; <information> may be zero or more <kw>=<value> pairs
; Block=<blockname specifies the block associated with this memory cell.
; Latch=<name> specifies the latch associated with this memory cell.
; Net=<netname> specifies the user net associated with this memory cell.
; COMPARE=[YES | NO] specifies whether or not it is appropriate to compare this bit position between a
;      "program" and a "readback" bitstream.  If not present the default is NO.
; Ram=<ram id>:<bit> This is used in cases where a CLB function
; Rom=<ram id>:<bit> generator is used as RAM (or ROM). <Ram id> will be either 'F', 'G', or 'M', indicating
;      that it is part of a single F or G function generator used as RAM, or as a single RAM
;      (or ROM) built from both F and G. <Bit> is a decimal number.
; Info lines have the following form:
; Info <name>=<value> specifies a bit associated with the LCA configuration options, and the value of
;      that bit.  The names of these bits may have special meaning to software reading the .ll file.
Info STARTSEL0=1
Info Persist=1
Info Readback=Used
Bit 60819 0x00024e00 883 Block=CLB_R49C49.S0 Latch=YQ Net=register_file/r25<6>
Bit 60909 0x00024e00 973 Block=CLB_R54C49.S0 Latch=YQ Net=register_file/r16<4>
Bit 68307 0x00025a00 883 Block=CLB_R49C49.S0 Latch=XQ Net=register_file/r25<7>
Bit 68397 0x00025a00 973 Block=CLB_R54C49.S0 Latch=XQ Net=register_file/r16<5>
Bit 68415 0x00025a00 991 Block=CLB_R55C49.S0 Latch=XQ Net=register_file/r25<5>
Bit 68433 0x00025a00 1009 Block=CLB_R56C49.S0 Latch=XQ Net=register_file/r16<3>
Bit 82089 0x00041000 937 Block=CLB_R52C48.S1 Latch=YQ Net=status_register/value<1>
Bit 120723 0x00044e00 883 Block=CLB_R49C48.S0 Latch=YQ Net=register_file/r16<6>
Bit 120777 0x00044e00 937 Block=CLB_R52C48.S0 Latch=YQ Net=register_file/r16<0>
...
Bit 5892618 0x00c46400 778 Block=AA32 Latch=I Net=port_a_reg/N24840
Bit 5892647 0x00c46400 807 Block=AA30 Latch=I Net=port_b_reg/N24818
Bit 5892654 0x00c46400 814 Block=AA29 Latch=I Net=port_c_reg/N24856
Bit 5892655 0x00c46400 815 Block=AB31 Latch=I Net=instruction_register/inst_in<11>
Bit 5892683 0x00c46400 843 Block=AB30 Latch=I Net=instruction_register/inst_in<10>
...
Bit 6044186 0x02020000 90 Block=RAMB4_R1C1 Ram=B:BIT63
Bit 6044187 0x02020000 91 Block=RAMB4_R1C1 Ram=B:BIT31
Bit 6044188 0x02020000 92 Block=RAMB4_R1C1 Ram=B:BIT30
Bit 6044191 0x02020000 95 Block=RAMB4_R1C1 Ram=B:BIT22
...
```

A.3 User Constraints file

The User Constraints file [119] is usually required to specify different constraints in the implementation of the design. They include temporal constraints, like the maximum acceptable delay of a certain path, and allocation constraints, like the mapping of a number of model signals to certain pins of the FPGA.

Table A.3 shows a sample User Constraints file for a Virtex FPGA.

Comments begin with '#'. For our purposes, lines beginning with the keyword 'NET' can also be ignored. They indicate the pin that must be associated to a particular signal of the model. For instance, the signal 'sram0_we' must drive the pin 'AJ25' of the FPGA, which connects to the write enable pin of the memory bank 0 of the prototyping board.

Lines beginning with the keyword 'INST' locate resources associated to model signals. This information is later compared with the LUTs used to implement the system's model in order to determine the location and name of each particular signal driven by a LUT.

Table A.3: *Sample User Constraints File (.ucf).*

```
# Start of Constraints extracted by Floorplanner from the Design
INST "clkg" LOC = "GCLKBUF3" ;
NET "sram0_we" LOC = "AJ25" ;
NET "sram0_oe" LOC = "AK26" ;
NET "sram0_cs0" LOC = "AN31" ;
NET "sram0_addr<19>" LOC = "AM31" ;
NET "sram0_addr<18>" LOC = "AD29" ;
...
INST "status_register/I8_0" LOC = "TBUF_R42C44.0" ;
INST "status_register/I8_1" LOC = "TBUF_R43C40.0" ;
INST "status_register/I8_2" LOC = "TBUF_R44C37.0" ;
...
INST "controller/Ker355651" LOC = "CLB_R44C35.S0" ;
INST "controller/Ker355581" LOC = "CLB_R42C35.S1" ;
INST "program_counter/_n003740" LOC = "CLB_R51C30.S0" ;
INST "program_counter/_n003640" LOC = "CLB_R52C34.S1" ;
INST "program_counter/_n003540" LOC = "CLB_R51C24.S1" ;
INST "register_file/Mmux__n0069_Result<3>1" LOC = "CLB_R55C75.S1" ;
INST "register_file/Mmux__n0069_Result<2>1" LOC = "CLB_R55C75.S1" ;
...
```

A.4 Workload file

The workload file specifies, in a custom format, the contents of the memory blocks of either the FPGA or the prototyping board. These memories will be used to provide the implemented system with the workload it must execute during the experimentation.

Table A.4 details the structure of a sample workload file.

In this example, the workload is going to be downloaded into the prototyping board memory blocks. Data must be stored as 32-bits words in big endian format. After the actual data, a comment (beginning with '//') can be added for explanation.

Table A.4: *Sample workload file for the prototyping board's external memory.*

```
0x0A 0x0C 0x00 0x00 // 0000:  movlw 0x0A
0x28 0x00 0x00 0x00 // 0001:  movwf R8
0x00 0x0C 0x00 0x00 // 0002:  movlw 0x00
0xC8 0x01 0x00 0x00 // 0003:  addwf R8,w
0xE8 0x02 0x00 0x00 // 0004:  decfsz R8
0x03 0x0A 0x00 0x00 // 0005:  goto 0x03
...
```

A.5 Readback command

A readback command is a particular bistream file that retrieves some information from the FPGA's configuration memory. A number of words from a particular position of the device's configuration memory are sent to the host.

An example of a *readback* command is shown in Table A.5.

In this case 142902 words are sent, starting from the memory position associated to the first CB (located in row 0 and column 0).

Table A.5: *Analysis of a* readback *command.*

Bitstream data	Description
AA995566	Synchronisation word
30002001	Packet Header: Write to FAR (Frame Address Register)
00000000	Packet Data: Frame Address (CLB, 0, 0)
30008001	Packet Header: Write to CMD (CoMmanD) register
00000004	Packet Data: RCFG (Read ConFiGuration data)
28006000	Packet Header: Write to FDRO (Frame Data Register Output)
48022E36	Packet Header Type 2: Data words (142902 words)
00000000	Dummy word

A.6 Trace file

Trace files store the observations perform during the experiments execution in a custom format.

An example of a fault-free execution (Golden Run) trace file is presented in Table A.6.

The Golden Run trace file presents a series of lines, beginning with '%', that specify the whole set of observation points that have been monitored.

The rest of the lines show, in hexadecimal format, the current state of these sequential elements at the time the observation was performed. This time, expressed in clock cycles, is stated at the beginning of the line. The contents of memory banks, preceded by the keyword 'Bank', follow the same format.

Table A.6: *Sample trace file of a Golden Run execution.*

```
% rom_data[7,6,5,4,3,2,1,0]
% U_CTR/alu_src_0[2]
% U_RAM/dout[7,6,5,4,3,2,1,0]
...
% alu_op_code[3,2,1,0]
% MemoryBank 1 60 words
...
13120 :  0 0 1 0 ff 0 1 1 ...
Bank 1 :  00030 0000 0001 ...
...
```

An example of the trace file of a fault injection experiment is presented in Table A.7.

In this case, the specification of the observation points is not included, since it is the same as in the Golden Run. Instead, some information regarding the injected fault is included (preceded by '--').

Observations follow the same format as in the Golden Run.

Table A.7: *Sample trace file of a workload's execution in presence of faults.*

```
-- CLB(23,49).SO_YQ U_RAM/dout(1)
...
13120 :  0 0 10 0 1 0 ff ...
Bank 1 :  00030 0000 0001 ...
...
```

A.7 Results file

An example of the file that stores the measures obtained from the analysis of the trace files of a fault injection experiment is presented in Table A.8.

Each line of this file states the considered observation point, the number and percentage of experiments with a trace equal to the Golden Run trace (system not affected by the fault), the number and percentage of experiments in which differences with the Golden Run trace do not involve those observation points selected as most significative (system moderately affected by the fault), and finally, the number and percentage of experiments with differences in significative observation points with respect to the Golden Run trace (system severely affected by the fault).

The last line shows the global results for the whole set of experiments.

Table A.8: *Sample results file.*

```
U_CTR/reg_pc_7_0 14 29.78 6 12.76 27 57.44
alu_op_code 19 82.60 0 0.0 4 17.39
U_RAM/sfr_psw 19 42.22 17 37.77 9 20.0
...
U_CTR/reg_f0 4 80.0 1 20.0 0 0.0
TOTAL 909 50.66 738 41.13 147 8.19
```

Bibliography

[1] C. Constantinescu, "Impact of Deep Submicron Technology on Dependability of VLSI Circuits," in *IEEE International Conference on Dependable Systems and Networks (DSN'02)*, (Bethesda, USA), pp. 205–209, 2002.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[3] J. Arlat and Y. Crouzet, "Faultload Representativeness for Dependability Benchmarking," in *IEEE International Symposium on Dependable Systems and Networks (DSN) - Workshop on Dependability Benchmarking*, (Washington, USA), pp. F29–F30, 2002.

[4] P. Gil, J. Arlat, H. Madeira, Y. Crouzet, T. Jarboui, K. Kanoun, T. Marteau, J. Durães, M. Vieira, D. Gil, J. C. Baraza, and J. Gracia, "Fault Representativeness," tech. rep., Deliverable ETIE2, Report from the "Dependability Benchmarking" Project (IST-2000-25425), 2002.

[5] T. Karnik, P. Hazucha, and J. Patel, "Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 128–143, 2004.

[6] E. Jenn, *Sur la validation des systèmes tolérant les fautes: injection de fautes dans des modèles de simulation VHDL.* PhD thesis, Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS, France, 1994.

[7] D. Gil, *Validación de sistemas tolerantes a fallos mediante inyección de fallos en modelos VHDL.* PhD thesis, Universidad Politécnica de Valencia, Spain, 1999.

[8] K. Compton, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[9] S. Hauck, *Multi-FPGA Systems*. PhD thesis, University of Washington, USA, 1995.

[10] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems," *Proceedings of the IEEE*, vol. 86, no. 4, pp. 615–638, 1998.

[11] U. R. Khan, H. L. Owen, and J. L. A. Hughes, "FPGA Architectures for ASIC Hardware Emulators," in *Sixth Annual IEEE International ASIC Conference*, (New York, USA), pp. 336–340, 1993.

[12] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, E. Martins, D. Powell, J.-C. Fabre, and J.-C. Laprie, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[13] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[14] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of Physical and Software-Implemented Fault Injection Techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[15] H. Madeira, M. Rela, F. Moreira, and J. G. Silva, "RIFLE: A General Purpose Pin-level Fault Injector," in *First European Dependable Computing Conference (EDCC-1)* (K. Echtle, D. Hammer, and D. Powell, eds.), (Budapest, Hungary), pp. 199–216, Springer-Verlag, 1994.

[16] R. J. Martínez, P. J. Gil, G. Martín, C. Pérez, and J. J. Serrano, "Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection," in *Seventh IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7)*, (San José, USA), pp. 249–265, 1999.

[17] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, and G. Leber, *Integration and Comparison of Three Physical Fault Injection Techniques*, ch. 5, pp. 309–329. Springer Verlag, 1995.

[18] J. Gaisler, "A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture," in *International Conference on Dependable Systems and Networks (DSN2002)*, (Washington, USA), pp. 409–415, 2002.

[19] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation," in *19th*

IEEE International Symposium on Fault Tolerant Computing (FTCS-19), (Chicago, USA), pp. 340–347, 1989.

[20] P. Folkesson, S. Svensson, and J. Karlsson, "A Comparison of Simulation Based and Scan Chain Implemented Fault Injection," in *International Symposium on Fault-Tolerant Computing (FTCS)*, (Munich, Germany), pp. 284–293, 1998.

[21] R. Velazco, C. Bellon, and B. Martinet, "Failure coverage of functional test methods: a comparative experimental evaluation," in *IEEE International Test Conference (ITC)*, (Washington, USA), pp. 1012–1017, 1990.

[22] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, 1995.

[23] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125–136, 1998.

[24] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles, "Dependability of COTS Microkernel-Based Systems," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 138–163, 2002.

[25] P. Yuste, J. Ruiz, L. Lemus, and P. Gil, "Non-intrusive Software Implemented Fault Injection in Embedded Systems," in *1st Latin-American Symposium on Dependable Computing (LADC)* (R. de Lemos and T. S. Weber and J. B. Camargo Jr., ed.), (Sao Paulo, Brazil), pp. 23–38, Springer-Verlag Berlin Heidelberg, 2003.

[26] IEEE-ISTO 5001-1999, *The Nexus 5001 Forum^{TM}Standard for a Global Embedded Processor Debug Interface.*, 1999.

[27] Dependability Benchmarking. http://www2.laas.fr/DBench/index.html. European Commission, IST-2000-25425, 2003.

[28] J. C. Baraza, *Contribución a la validación de sistemas complejos tolerantes a fallos en la fase de diseño. Nuevos modelos de fallos y técnicas de inyección de fallos.* PhD thesis, Universidad Politécnica de Valencia, Spain, 2003.

[29] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool," in *24th International*

Symposium on Fault-Tolerant Computing (FTCS-24), (Austin, USA), pp. 66–75, 1994.

[30] V. Sieh, O. Tschäche, and F. Balbach, "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions," in *27th International Symposium on Fault Tolerant Computing (FTCS-27)*, (Seattle, USA), pp. 32–36, 1997.

[31] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, "A prototype of a VHDL-based fault injection tool: description and application," *Journal of Systems Architecture: the EUROMICRO Journal*, vol. 47, no. 10, pp. 847–867, 2002.

[32] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[33] IEEE, *Verilog Hardware Description Language*, 2005. IEEE Standard No. 1364-2005.

[34] IEEE, *Behavioural languages–Part 1: VHDL language reference manual*, 2004. IEEE Standard No. 61691-1-1: 2004.

[35] IEEE, *SystemC Language Reference Manual*, 2005. IEEE Standard No. 1666-2005.

[36] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[37] R. A. Rutenbar, "Simulated Annealing Algorithms: An Overview," *IEEE Circuits and Devices Magazine*, vol. 5, no. 1, pp. 19–26, 1989.

[38] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, and O. Lepape, "Serial Fault Emulation," in *Design Automation Conference (DAC)*, (Las Vegas, USA), pp. 801–806, 1996.

[39] B. L. Hutchings and M. J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications," in *International Workshop on Field Programmable Logic and Applications (FPL)*, (Oxford, UK), pp. 293–302, 1995.

[40] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai, "Fault Emulation: A New Approach to Fault Grading," in *International Conference on Computer-Aided Design (ICCAD)*, (San José, USA), pp. 681–686, 1995.

[41] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai, "Fault Emulation: A New Methodology for Fault Grading," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (CADICS)*, vol. 18, no. 10, pp. 1487–1495, 1999.

[42] J.-H. Hong, S.-A. Hwang, and C.-W. Wu, "An FPGA-based Hardware Emulator for Fast Fault Emulation," in *IEEE 39th Midwest Symposium on Circuits and Systems*, vol. 1, (Ames, USA), pp. 345–348, 1996.

[43] S.-A. Hwang, J.-H. Hong, and C.-W. Wu, "Sequential Circuit Fault Simulation Using Logic Emulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (CADICS)*, vol. 17, no. 8, pp. 724–736, 1998.

[44] R. Sedaghat-Maman, *Fault Emulation: Reconfigurable Hardware-Based Fault Simulation Using Logic Emulation Systems with Optimized Mapping*. PhD thesis, Universität Hannover, Germany, 1999.

[45] M. B. Santos, I. M. Teixeira, and J. P. Teixeira, "Dynamic Fault Injection Optimization for FPGA-Based Hardware Fault Simulation," in *IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, (Brno, Czech Republic), pp. 370–373, 2002.

[46] R. Velazco, R. Leveugle, and O. Calvo, "Upset-like Fault Injection in VHDL Descriptions: A method and Preliminary Results," tech. rep., Techniques of Informatics and Microelectronics for computer Architeture (TIMA) Laboratory, 2001.

[47] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "Exploiting Circuit Emulation for Fast Hardness Evaluation," *IEEE Transactions on Nuclear Science (CADICS)*, vol. 48, no. 6, pp. 2210–2216, 2001.

[48] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "FPGA-based Fault Injection for Microprocessor Systems," in *10th Asian Test Symposium (ATS)*, (Kyoto, Japan), pp. 304–309, 2001.

[49] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "Exploiting FPGA-based Techniques for Fault Injection Campaigns on VLSI Circuits," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, (San Francisco, USA), pp. 250–258, 2001.

[50] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "An FPGA-Based Approach for Speeding-Up Fault Injection Campaigns on Safety-Critical Circuits," *Journal of Electronic Testing: Theory and Applications*, no. 18, pp. 261–271, 2002.

[51] F. Corno, M. S. Reorda, and G. Squillero, "RT-Level ITC 99 Benchmarks and First ATPG Results," *IEEE Design and Test of Computers*, vol. 17, no. 3, pp. 44–53, 2000.

[52] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "New Techniques for efficiently assessing reliability of SOCs," *Microelectronics Journal*, vol. 34, no. 1, pp. 53–61, 2003.

[53] C. López-Ongil, M. García-Valderas, M. Portela-García, and L. Entrena, "An autonomous FPGA-based emulation system for fast fault tolerant evaluation," in *International Conference on Field-Programmable Logic and Applications (FPL)*, (Tampere, Finland), pp. 397–402, 2005.

[54] C. López-Ongil, M. García-Valderas, M. Portela-García, and L. Entrena, "Autonomous Fault Emulation: A New FPGA-Based Acceleration System for Hardness Evaluation," *IEEE Transactions on Nuclear Science*, vol. 54, no. 1, pp. 252–261, 2007.

[55] A. Ejlali, S. G. Miremadi, H. Zarandi, G. Asadi, and S. B. Sarmadi, "A Hybrid Fault Injection Approach Based on Simulation and Emulation Co-operation," in *International Conference on Dependable Systems and Networks (DSN)*, (San Francisco, USA), pp. 479–488, 2003.

[56] A. Ejlali and S. G. Miremadi, "FPGA-Based Fault Injection into Switch-Level Models," *Microprocessors and Microsystems*, vol. 28, no. 5–6, pp. 317–327, 2004.

[57] L. Antoni, R. Leveugle, and B. Fehér, "Using run-time reconfiguration for fault injection applications," in *IEEE Instrumentation and Measurement Technology Conference (IMTC)*, (Budapest, Hungary), pp. 1773–1777, 2001.

[58] L. Antoni, R. Leveugle, and B. Fehér, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, (San Francisco, USA), pp. 403–413, 2001.

[59] L. Antoni, R. Leveugle, and B. Fehér, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes," in *IEEE International Sym-*

posium on Defect and Fault Tolerance in VLSI Systems (DFT), (Vancouver, Canada), pp. 245–253, 2002.

[60] L. Antoni, R. Leveugle, and B. Fehér, "Using Run-Time Reconfiguration for Fault Injection Applications," *IEEE Transactions on Instrumentation and Measurement*, vol. 52, no. 5, pp. 1468–1473, 2003.

[61] R. Leveugle and L. Antoni, "Dependability Analisys: a New Application for Run-Time Reconfiguration," in *International Parallel and Distributed Processing Symposium (IPDPS)*, (Nice, France), pp. 173–179, 2003.

[62] L. Antoni, *Injection de Fautes par Reconfiguration Dynamique de Réseaux Programmables*. PhD thesis, Institut National Polytechnique de Grenoble, France, 2003.

[63] M. Aguirre, J. N. Tombs, F. Muñoz, V. Baena, A. Torralba, A. Fernández-León, F. Tortosa, and D. González-Gutiérrez, "An FPGA based hardware emulator for the insertion and analysis of Single Event Upsets in VLSI Designs," in *Radiation Effects on Components and Systems Workshop (RADECS)*, (Madrid, Spain), pp. 1–5, 2004.

[64] A. Parreira, J. P. Teixeira, and M. B. Santos, "Built-In Self-Test Preparation in FPGAs," in *7th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, (Tatrmaská Lomnica, Slovakia), p. 8, 2004.

[65] A. Parreira, J. P. Teixeira, and M. B. Santos, "FPGAs BIST Evaluation," in *International Conference on Field-Programmable Logic and its Applications (FPL)*, (Antwerp, Belgium), p. 10, 2004.

[66] A. Parreira, J. P. Teixeira, and M. B. Santos, "Built-In Self-Test Quality Assessment Using Hardware Fault Emulation In FPGAs," *Computing and Informatics*, vol. 23, no. 5-6, pp. 1001–1020, 2004.

[67] F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profile of Sequential Benchmark Circuits," in *International Symposium on Circuits and Systems (ISCAS)*, (Portland, USA), pp. 1229–1234, 1989.

[68] M. Böhnel and R. Weiss, "Self-Stabilization of LUT-Based FPGA Designs by Fault Injection," in *7th IEEE International On-Line Testing Workshop (IOLTW)*, (Taormina, Italy), p. 139, 2001.

[69] M. B. Tahoori and S. Mitra, "Techniques and Algorithms for Fault Grading of FPGA Interconnect Test Configurations," *IEEE Transaction on*

Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 2, pp. 261–272, 2004.

[70] F. G. de Lima, *Designing Single Event Upset Mitigation Techniques for Large SRAM-Based FPGA Components*. PhD thesis, Universidade Federal do Rio Grande do Sul, Brazil, 2003.

[71] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new functional model for FPGA Application-Oriented testing," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, (Vancouver, Canada), pp. 372–380, 2002.

[72] Xilinx Inc, *Virtex^{TM}2.5 V Field Programmable Gate Arrays: Functional Description*, 2002. DS003-2 (v2.8.1).

[73] Xilinx Inc, *Virtex-II Platform FPGAs: Complete Data Sheet*, 2004. DS031 (v3.3).

[74] Xilinx Inc, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 2005. DS083 (v4.5).

[75] Xilinx Inc, *Virtex-4 Family Overview*, 2006. DS112 (v1.5).

[76] Xilinx Inc, *Virtex-5 LX Platform Overview*, 2006. DS100 (v1.1).

[77] Altera Corp, *Stratix Device Family Data Sheet, Volume 1*, 2006. ver 3.4.

[78] Altera Corp, *Stratix GX Device Family Data Sheet, Volume 1*, 2006. ver 1.2.

[79] Altera Corp, *Stratix II Device Family Data Sheet, Volume 1*, 2006. ver 4.1.

[80] Altera Corp, *Stratix II GX Device Family Data Sheet, Volume 1*, 2006. ver 3.0.

[81] Lattice Semiconductor Corporation, *LatticeSC Family Data Sheet*, 2006. Version 01.0.

[82] Atmel Corp, *AT40KAL Series FPGA*, 2004. 2818E-FPGA-1/04.

[83] D. de Andrés, J. C. Ruiz, D. Gil, and P. Gil, "Run-Time Reconfiguration for Emulating Transient Faults in VLSI Systems," in *International Conference on Dependable Systems and Networks (DSN)*, (Philadelphia, USA), pp. 291–300, 2006.

[84] D. de Andrés, J. C. Ruiz, D. Gil, and P. Gil, "Fast Emulation of Permanent Faults in VLSI Systems," in *International Conference on Field Programmable Logic and Applications (FPL)*, (Madrid, Spain), pp. 247–252, 2006.

[85] D. de Andrés, J. Albaladejo, L. Lemus, and P. Gil, "Fast Run-Time Reconfiguration for SEU Injection," in *Fifth European Dependable Computing Conference (EDCC-5)* (M. Cin, M. Kaâniche, and A. Pataricza, eds.), (Budapest, Hungary), pp. 230–245, Springer-Verlag Berlin Heidelberg, 2005.

[86] A. Parreira, J. P. Teixeira, A. Pantelimon, M. B. Santos, and J. T. de Sousa, "Fault Simulation using Partially Reconfigurable Hardware," in *International Conference on Field-Programmable Logic and its Applications (FPL)*, (Lisboa, Portugal), pp. 839–848, 2003.

[87] A. Parreira, J. P. Teixeira, and M. Santos, "A Novel Approach to FPGA-Based Hardware Fault Modeling and Simulation," in *6th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, (Poznan, Poland), pp. 17–24, 2003.

[88] W. Quine, "A way to simplify truth functions," *American Mathematical Monthly*, vol. 62, pp. 627–631, 1955.

[89] E. J. McCluskey, "Minimization of Boolean Functions," *Bell Systems Technical Journal*, vol. 35, pp. 1417–1444, 1956.

[90] Xilinx Inc, *Virtex^{TM}2.5 V FPGA DC and Switching Characteristics*, 2002. DS003-3 (v3.2).

[91] M. G. Gericota, G. R. Alves, M. L. Silva, and J. M. Ferreira, "On-line Testing of FPGA Logic Blocks Using Active Replication," in *Proceedings of the Norsk Informatikkonferanse (NIK'2002)*, (Kongsberg, Norway), pp. 167–178, 2002.

[92] I. Hadzic, S. Udani, and J. M. Smith, "Fpga viruses," in *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, (London, UK), pp. 291–300, Springer-Verlag, 1999.

[93] D. de Andrés, J. C. Ruiz, D. Gil, and P. Gil, "FADES: a Fault Emulation Tool for Fast Dependability Assessment," in *International Conference on Field Programmable Technology (FPT)*, (Bangkok, Thailand), pp. 221–228, 2006.

[94] Atmel Corp, *AT40K Series Configuration*, 2002. 1009B-FPGA-03/02.

[95] Atmel Corp, *Atmel FPGA Integrated Development System (IDS)*, 2001. 1421D-06/01.

[96] Atmel Corp, *FPSLIC on-chip Partial Reconfiguration of the Embedded AT40K FPGA*, 2002. 3013A-FPSLI-01/02.

[97] Xilinx Inc, *Virtex FPGA Series Configuration and Readback*, 2002. XAPP138 (v2.7).

[98] Xilinx Inc, *Virtex Series Configuration Architecture User Guide*, 2003. XAPP151 (v1.6).

[99] Xilinx Inc, *Xilinx ISE 8 Software Manuals and Help - PDF Collection*, 2006.

[100] Xilinx Inc, *PlanAhead User Guide*, 2006. Release 8.2.

[101] E. Lechner and S. A. Guccione, "The Java Environment for Reconfigurable Computing," in *International Workshop on Field-Programmable Logic and Applications (FPL)*, (Londres, UK), pp. 284–293, 1997.

[102] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing," in *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[103] S. McMillan and S. A. Guccione, "Partial Run-Time Reconfiguration Using JRTR," in *International Workshop on Field-Programmable Logic and Applications (FPL)*, (Villach, Austria), pp. 352–360, 2000.

[104] Xilinx Inc, *The JBits 2.8 SDK for Virtex*, 2001.

[105] Celoxica Inc, *RC1000 Functional Reference Manual*, 2001. RM-1140-0.

[106] E. Keller, "JRoute: A Run-Time Routing API for FPGA Hardware," in *IPDPS Workshops on Parallel and Distributed Processing* (J. D. P. Rolim, ed.), vol. 1800, (Cancun, Mexico), pp. 874–881, 2000.

[107] A. Benso and P. Prinetto, *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation.* Kluwer Academic Publishers, 2003.

[108] Oregano Systems, "MC8051 IP Core." http://www.oregano.at/en/ip/index.htm, 2006. version 1.5.

[109] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil, "Study, comparison and application of different VHDL-based fault injection techniques for the experimental validation of a fault-tolerant system," *Microelectronics Journal*, vol. 34, no. 1, pp. 41–51, 2003.

[110] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil, "Analysis of the influence of processor hidden registers on the accuracy of fault injection techniques," in *9th IEEE International High-Level Design Validation and Test Workshop*, (California, USA), pp. 173–178, 2004.

[111] D. Gil, J. Gracia, J. C. Baraza, and P. J. Gil, "Impact of Faults in Combinational Logic of Commercial Microcontrollers," *Lecture Notes in Computer Science*, pp. 379–390, 2005.

[112] P. Lidén, P. Dahlgren, R. Johansson, and J. Karlsson, "On Latching Probability of Particle Induced Transients in Combinational Networks," in *International Symposium on Fault-Tolerant Computing (FTCS-24)*, (Austin, USA), pp. 340–349, 1994.

[113] L. Chwif, M. R. P. Barretto, and R. J. Paul, "On Simulation Model Complexity," in *Proceedings of the 2000 Winter Simulation Conference*, (Orlando, USA), pp. 449–455, 2000.

[114] E. Romani, "Structural PIC165X microcontroller." Hamburg VHDL Archive, http://tams-www.informatik.uni-hamburg.de/vhdl/, 1998.

[115] D. de Andrés, J. C. Ruiz, D. Gil, and P. Gil, "Fault Emulation for Dependability Evaluation of VLSI Systems," *IEEE Transactions on Very Large Scale Integration Systems*, to be published, fourth quarter 2007.

[116] N. Kropp, P. Koopman, and D. Siewiorek, "Automated Robustness Testing of Off the Shelf Software Components," in *28th Fault Tolerant Computing Symposium (FTCS)*, (Munich, Germany), pp. 23–25, 1998.

[117] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[118] D. P. Siewiorek and R. S. Swarz, *Reliable Computer System - Design and Evaluation*. Digital Press, 1992.

[119] Xilinx Inc, *Constraints Guide*, 2007. 9.1i.