UNIVERSITÀ DI PISA
Scuola di Dottorato in Ingegneria "Leonardo da Vinci"

UNIVERSITÀ DI PISA

Corso di Dottorato di Ricerca in
INGEGNERIA DELL'INFORMAZIONE

Tesi di Dottorato di Ricerca

Analysis and Test of the Effects of Single Event Upsets Affecting the Configuration Memory of SRAM-based FPGAs

Luca Maria Cassano

Anno 2013

UNIVERSITÀ DI PISA

Scuola di Dottorato in Ingegneria "Leonardo da Vinci"

Corso di Dottorato di Ricerca in
INGEGNERIA DELL'INFORMAZIONE

Tesi di Dottorato di Ricerca

Analysis and Test of the Effects of Single Event Upsets Affecting the Configuration Memory of SRAM-based FPGAs

Autore:

Luca Maria Cassano _____

Relatori:

Prof. Cinzia Bernardeschi _____

Prof. Gianluca Dini _____

Ing. Andrea Domenici _____

Anno 2013
SSD ING-INF/05

Sommario

I dispositivi FPGA con memoria di configurazione SRAM sono sempre più rilevanti in un grande numero di campi applicativi, dal contesto automobilistico a quello aerospaziale. Questi campi applicativi sono caratterizzati dalla presenza di radiazioni capaci di causare Single Event Upsets (SEUs) in dispositivi digitali. Tali guasti hanno effetti particolarmente dannosi sui sistemi implementati in tecnologia SRAM-based FPGA, in quanto sono in grado non solo di danneggiare temporaneamente il comportamento del sistema, cambiando il contenuto di flip-flop e memorie, ma anche di cambiare permanentemente la funzionalità implementata dal sistema stesso, cambiando il contenuto della memoria di configurazione. Il design di applicazioni safety-critical richiede l'utilizzo, prima possibile durante il flusso di progetto del sistema, di metodologie accurate per la valutazione della sensitività ai SEU del sistema stesso. Inoltre è necessario essere in grado di rilevare l'occorrenza di SEU durante il funzionamento del sistema. A questo scopo è necessario generare test patterns durante il progetto del sistema ed è poi necessario applicare tali test patterns agli input del sistema durante il suo funzionamento.

In questa tesi descriviamo il progetto e l'implementazione di strumenti software utili al progettista di applicazioni safety-critical basati su tecnologia SRAM-based FPGA per la valutazione della sensitività ai SEU del sistema e per la generazione di test pattern utili al rilevamento di SEU nella memoria di configurazione durante la vita del sistema. La caratteristica principale di questi strumenti è l'implementazione di un modello di SEU nei bit di configurazione che controllano le risorse logiche e di routing di un dispositivo FPGA che risulta essere molto più accurato rispetto ai classici modelli stuck-at ed open/short che sono in genere considerati nell'analisi di circuiti digitali. In tal modo gli strumenti proposti risultano essere molto più accurati rispetto a strumenti simili, sia accademici che commerciali, attualmente disponibili per l'analisi dei guasti in dispositivi digitali ma non specificamente sviluppati per dispositivi FPGA.

In particolare tre strumenti sono stati progettati ed implementati: (i) *ASSESS: Accurate Simulator of SEuS affecting the configuration memory of SRAM-based FPGAs*, un simulatore di SEU nella memoria di configurazione di sistemi implementati in tec-

nologia SRAM-based FPGA, finalizzato a valutare la sensitività del sistema ai SEU prima possibile nel processo di sviluppo del sistema; (ii) *UA^{2}TPG: Untestability Analyzer and Automatic Test Pattern Generator for SEUs Affecting the Configuration Memory of SRAM-based FPGAs*, uno strumento di analisi statica per l'identificazione dei SEU non testabili e per la generazione automatica di test patterns per il rilevamento del 100% dei SEU testabili; e (iii) *GABES: Genetic Algorithm Based Environment for SEU Testing in SRAM-FPGAs*, un ambiente basato su un algoritmo genetico per la generazione ed ottimizzazione di test patterns per il rilevamento di SEU nella memoria di configurazione del sistema.

Gli strumenti proposti sono stati applicati ad alcuni circuiti del benchmark ITC'99. I risultati ottenuti da questi esperimenti sono stati confrontati con i risultati ottenuti da esperimenti simili, in cui abbiamo considerato quasti stuck-at anzichè il modello accurato di guasti SEU. Dal confronto fra questi esperimenti abbiamo potuto verificare che gli strumenti software proposti sono effettivamente più accurati rispetto a strumenti simili oggi disponibili. In particolare, il confronto fra i risultati ottenuti usando ASSESS e quelli ottenuti attraverso la fault injection ha riportato che il simulatore di guasti proposto ha un errore medio dello 0.1% ed un errore massimo dello 0.5%, mentre usando un simulatore di guasti basato sul modello stuck-at l'errore medio ottenuto rispetto alla fault injection è del 15.1% e l'errore massimo è del 56.2%. Analogamente il confronto fra i risultati ottenuti usando UA^{2}TPG per il modello accurato di SEU, con i risultati ottenuti per i guasti stuck-at ha rivelato una differenza media di untestability del 7.9% ed una massima del 37.4%. Infine il confronto fra i valori di fault coverage ottenuti dai test patterns generati per il modello accurato di SEU e le fault coverage ottenute dai test patterns generati per guasti stuck-at ha mostrato che mentre i primi coprono il 100% dei guasti testabili, i secondi coprono in media il 78.9% dei guasti testabili, con una copertura minima del 54% ed una copertura massima del 93.16%.

Abstract

SRAM-based FPGAs are increasingly relevant in a growing number of safety-critical application fields, ranging from automotive to aerospace. These application fields are characterized by a harsh radiation environment that can cause the occurrence of Single Event Upsets (SEUs) in digital devices. These faults have particularly adverse effects on SRAM-based FPGA systems because not only can they temporarily affect the behaviour of the system by changing the contents of flip-flops or memories, but they can also permanently change the functionality implemented by the system itself, by changing the content of the configuration memory. Designing safety-critical applications requires accurate methodologies to evaluate the system's sensitivity to SEUs as early as possible during the design process. Moreover it is necessary to detect the occurrence of SEUs during the system life-time. To this purpose test patterns should be generated during the design process, and then applied to the inputs of the system during its operation.

In this thesis we propose a set of software tools that could be used by designers of SRAM-based FPGA safety-critical applications to assess the sensitivity to SEUs of the system and to generate test patterns for in-service testing. The main feature of these tools is that they implement a model of SEUs affecting the configuration bits controlling the logic and routing resources of an FPGA device that has been demonstrated to be much more accurate than the classical stuck-at and open/short models, that are commonly used in the analysis of faults in digital devices. By keeping this accurate fault model into account, the proposed tools are more accurate than similar academic and commercial tools today available for the analysis of faults in digital circuits, that do not take into account the features of the FPGA technology..

In particular three tools have been designed and developed: (i) *ASSESS: Accurate Simulator of SEuS affecting the configuration memory of SRAM-based FPGAs*, a simulator of SEUs affecting the configuration memory of an SRAM-based FPGA system for the early assessment of the sensitivity to SEUs; (ii) *UA^{2}TPG: Untestability Analyzer and Automatic Test Pattern Generator for SEUs Affecting the Configuration Memory of SRAM-based FPGAs*, a static analysis tool for the identification of the untestable

SEUs and for the automatic generation of test patterns for in-service testing of the 100% of the testable SEUs; and (iii) *GABES: Genetic Algorithm Based Environment for SEU Testing in SRAM-FPGAs*, a Genetic Algorithm-based Environment for the generation of an optimized set of test patterns for in-service testing of SEUs.

The proposed tools have been applied to some circuits from the ITC'99 benchmark. The results obtained from these experiments have been compared with results obtained by similar experiments in which we considered the stuck-at fault model, instead of the more accurate model for SEUs. From the comparison of these experiments we have been able to verify that the proposed software tools are actually more accurate than similar tools today available. In particular the comparison between results obtained using ASSESS with those obtained by fault injection has shown that the proposed fault simulator has an average error of 0.1% and a maximum error of 0.5%, while using a stuck-at fault simulator the average error with respect of the fault injection experiment has been 15.1% with a maximum error of 56.2%. Similarly the comparison between the results obtained using UA^{2}TPG for the accurate SEU model, with the results obtained for stuck-at faults has shown an average difference of untestability of 7.9% with a maximum of 37.4%. Finally the comparison between fault coverages obtained by test patterns generated for the accurate model of SEUs and the fault coverages obtained by test pattern designed for stuck-at faults, shows that the former detect the 100% of the testable faults, while the latter reach an average fault coverage of 78.9%, with a minimum of 54% and a maximum of 93.16%.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

Since the first FPGA device, the XC2064 [195], was developed by Xilinx in 1985 the FPGA technology has enormously grown in terms of flexibility, reliability and computational power. Although it is still not comparable with the Application Specific Integrated Circuit (ASIC) technology either in terms of computational power or of silicon area occupation [114], the FPGA technology has imposed itself both in safety critical and in non safety critical application fields thanks to very good performance, low non recurrent design cost and very short time to market.

In the last years SRAM-based FPGAs have increasingly been employed in safety-related applications such as railway signaling [40], nuclear power plant I&C [173], radar systems for automotive applications [190], aerospace [99, 110], and avionics [116].

Given their relevance, and the severity of the occurring of accidents, the design of both hardware and software safety critical systems is regulated by application field specific standards, such as the ISO 26262-5 [106] and the ISO 26262-6 [107] for automotive applications, the CENELEC 50128 [69] and the CENELEC 50129 [70] for railway applications, the IAEA-NS-G-1.1 [104] and the IAEA NS-G-1.3 [105] for software and hardware control systems for nuclear power plants, the Q-ST-60-02C [77] for aerospace applications, the DO-178B [75] and the DO-254 [76] for avionic applications. These standards state which activities are mandatory, which ones are recommended, and which ones are optional and which techniques, methods and tools have to be used for each phase of the design cycle.

Nowadays an official regulation specific for the FPGA design in safety-critical systems only exists in the aerospace application field [77]. Other guidelines and standards for the design and verification of programmable devices have been developed internally to companies [1, 73, 95, 175]. In any other safety-critical application fields regulations exist for the development of software and hardware (ASICs, boards, microcontrollers), but a regulation that keeps in consideration all those features specific of the FPGA technology does not exist. Thus the existing verification and validation

approaches only partially satisfy the requirements for the verification and validation of safety-critical FPGA-based systems.

In [45] the necessity of adopting verification and validation methodologies keeping into account the peculiarities of the FPGA technology is strongly endorsed because often ASIC-specific methodologies, procedures and tools are not adequate to the design of FPGA-based systems. Habinc *et al.* in [73] and [95], as well as Gibbons and Ames in [82], discuss how many problems and failures in space applications involving FPGA devices are the result of applying inadequate development, verification and validation methodologies.

The early assessment of the sensitivity to run-time faults of the system and the detection of run-time faults during the system life-time are aspects considered as critical by both design standards, such as [70, 105, 106], and industrial technical reports [73, 95].

A significant source of run-time faults in SRAM-based FPGA system are radiations. Radiations in the atmosphere are responsible for introducing a number of disruptive effects in digital devices [27]. Among the various effects induced by radiations, *Single Event Upsets (SEUs)* have particularly adverse effects on FPGAs using SRAM technology since they are able not only to cause transient faults by changing the content of flip-flops, but they may also permanently corrupt a bit in the configuration memory (correctable only with a reconfiguration of the device) [89]. Such faults are not transient in FPGAs because the configuration memory is usually not written again after the first configuration. SEUs in the configuration memory bits disrupt the routing architecture of the implemented circuit by modifying the interconnections among components of the netlist and change the behaviour of the functional units by modifying the implemented functionalities.

For the assessment of the sensitivity to SEUs of an SRAM-based FPGA system four families of techniques exist: Radiation testing, fault injection, static analysis and fault simulation. Among the techniques for the evaluation of the effects of faults in digital circuits, fault simulation, such as [41, 43, 94, 161, 172], represents an interesting solution because it allows to assess the sensitivity to faults of the system under analysis early during the design process, before hardware prototypes are available. Thus fault simulation can represent the first and early analysis in a multi-step assessment of the sensitivity to faults of a systems, that should be concluded by prototype-based experiments, such as radiation testing and fault injection. Unfortunately no fault simulators that specifically take into account the effects of SEUs in the configuration memory of SRAM-based FPGAs today exist.

Many works addressing the problem of automatic test pattern generation (ATPG) for digital circuits have been published [4, 142], but very few of these works specifically address FPGAs. Test methods devised for ASIC circuits could be effective when used for testing structural defects in the FPGA chip, but they are not satisfactory when used for testing SEUs in the configuration memory of FPGAs [156]. In particular, it has been demonstrated that test pattern generation methods based on the stuck-at fault model

for ASIC circuits obtain too optimistic results when applied to SRAM-FPGAs. The stuck-at fault model considers permanent faults at the input and output terminals of the logical components. More accurate fault models, keeping into account faults in the configuration bits of the FPGA chip, should be considered [155].

The existing testing techniques for FPGA circuits can be classified in two main families: *application-independent* and *application-dependent*. Application-independent methods [102, 157, 179] aim at detecting structural defects due to the manufacturing process of the FPGA chip and they are performed by the chip manufacturer. These methods are called application-independent because they target every possible fault in the device without any consideration of which parts of the chip are actually used by the given design and which parts are not. Since these techniques generally use multiple re-configurations of the device, they cannot be employed for testing SEUs in the configuration memory. Conversely, application-dependent methods [30, 163, 180] address only the resources of the FPGA chip actually used by the implemented system. Given this, application-dependent methods can be used for in-service testing of both structural defects and SEUs. As with fault simulation, also in the field of fault testing no techniques exist that keep into account SEUs in the configuration memory of SRAM-based FPGAs, apart from the one proposed in [30].

Automatic test pattern generation (ATPG) for integrated circuits is a very hard task, since in modern *Very Large Scale of Integration (VLSI)* systems the total number of faults that need to be detected may be very large. This is particularly true for FPGA-based systems because not only faults in user resources, but also faults in the configuration memory of the device have to be detected. However a number of these faults may be demonstrated to be untestable, thus reducing the effort required of ATPG tools. Moreover, demonstrating the untestability of faults in a VLSI design offers the designer an evaluation of the degree of testability of the system. Like what we have previously discussed about test pattern generation, also for the untestability analysis problem it is true that while the stuck-at fault model could be considered when hardware defects in the FPGA device are addressed, this fault model is not accurate when SEUs in the configuration memory of an FPGA-based system have to be analysed. Works addressing the problem of demonstrating the untestability of stuck-at faults in digital circuits can be found in the literature [153, 151, 152, 182, 125], but no one specifically addresses the analysis of the testability of SEUs affecting the configuration memory of FPGA-based systems, apart from the ones presented by the author of the present dissertation in [33, 36], where the analysis of the excitability of SEUs is addressed.

1.2 Contribution of the Thesis

Very much work has been done in the field of radiation testing and fault injection for the analysis of the effects of SEUs. Similarly many works about the generation of test configurations for post-production detection of defects in FPGA devices can be

found in the literature. Nevertheless, the state of the art in the field of fault testing and analysis for SRAM-based FPGA systems lacks accurate software tools for the analysis of the effects of SEUs in the configuration memory of the device and for the generation of test patterns for in-service testing of such faults.

In this dissertation we present a framework of software tools for the analysis and test of SEUs affecting the configuration memory of SRAM-based FPGA systems during the life-time of the system. The framework is composed of a simulator of SEUs (*ASSESS*) for the assessment of the sensitivity to SEUs and of the failure probability due to SEUs, an untestability analyzer and test pattern generator (*UA*^{2}*TPG*) and a genetic algorithm-based environment (*GABES*) for the generation and optimization of test patterns for on-line testing.

ASSESS is an accurate simulator of SEUs affecting the configuration memory of SRAM-based FPGA systems. The simulator relies on a general model of FPGA circuits considered at the netlist level. The model is based on the formalism of *Stochastic Activity Networks* (SAN) [170] and it has been developed with the Möbius tool [49]. The simulator is able to emulate the effects of SEUs affecting any of the configuration bits actually used by a given design. In particular the simulator is able to accurately reproduce the modification of the functionalities implemented by LUTS induced by SEUs in configuration bits controlling logic components and the changes of the connections among components of the netlist induced by SEUs in configuration bits controlling routing components. ASSESS can be used for an early assessment of the sensitivity to SEUs of the system under design and for an estimation of the failure probability of the system. Moreover the SEU simulator can be used to assess the fault coverage of pre-generated test patterns.

UA^{2}TPG is an untestability prover and automatic test pattern generator for SEUs in the configuration memory of SRAM-based FPGA systems. The tool statically determines which SEUs in the configuration bits actually used by a given system are not testable. Moreover, at the end of the untestability analysis, the tool generates a set of test patterns able to detect 100% of the testable SEUs. The proposed tool addresses only the configuration bits actually used by the system under analysis, thus the generated test patterns may be used for application-dependent in-service testing. Moreover, if we look at the analysis performed by the tool from the fault tolerance point of view instead of the testing point of view, we can say that the tool assesses the sensitivity to SEUs of SRAM-based FPGA systems and generates input patterns that can be used to stimulate the system during fault injection or radiation testing experiments. The proposed tool relies on the SAL [29] description language to describe the structure of the netlist under analysis and to specify untestability theorems through LTL formulas [162]. The SAL-SMC model checker is used to prove the untestability of faults and to express the counter-examples that are used to generate the test patterns for the testable faults.

GABES is a genetic algorithm-based environment for SEU testing: A tool for automatic test pattern generation based on a *genetic algorithm* (GA) for application-

dependent in-service testing of SEUs in SRAM-FPGAs, that takes into account SEUs in configuration bits of the FPGA. The proposed GA uses ASSESS to calculate the fault coverage obtained by each generated test pattern. The main goal of GABES is producing efficient sets of test patterns for in-service testing, that can be optimized with respect either to fault coverage or to test speed, according to the specific application requirements. Another goal is optimizing the test pattern generation itself: even if the off-service process of test pattern generation is not subject to the stringent constraints of in-service testing, excessive computation times can make the method impractical.

All the discussed tools implement the fault model for SEUs affecting the configuration bits controlling logic resources proposed in [155] and the model for SEUs in configuration bits controlling routing resources proposed in [186]. These fault models have been demonstrated to be much more accurate than the classical stuck-at model (for logic resources) and the open/short model (for routing resources) when the analysis and test of SEUs in the configuration memory of SRAM-based FPGAs is addressed.

All the tools work in conjunction with an EDIF parser [109] and the *E*^{2}*STAR* tool [37]. The parser is able to translate the EDIF description of the netlist into an intermediate description of the topology of the netlist in terms of connections among logic components and functionalities performed by components. Moreover the parser is able to produce a list of the effects of SEUs occurring in configuration bits associated with the logic resources used by the system under analysis. These faults are represented in terms of the induced modification of the truth table of the affected LUT. *E*^{2}*STAR* is a static analyzer of the configuration memory of the SRAM-based FPGA device developed at the Politecnico di Torino. Given an FPGA device and a placed-and-routed design, *E*^{2}*STAR* is able to determine which are the configuration bits actually used by the design and which are the logical effects of SEUs occurring in the configuration bits controlling routing resources according to the fault model previously presented. For each configuration bit associated with a routing component *E*^{2}*STAR* reports the number of propagation points of an SEU occurring in the configuration bit, and for each propagation point *E*^{2}*STAR* reports the logical effect, the affected component(s) and the pins of the affected component(s) to which the fault propagates. Thanks to these two tools the proposed environment is fully integrated in the standard design process of FPGA-based systems since the input of the parser is the EDIF file produced by the synthesis tool and since the *E*^{2}*STAR* works directly on the post place-and-route netlist description file.

The main contribution of this thesis is to present the first simulator, untestability analyzer and automatic test pattern generator specifically intended to address SEUs affecting the configuration memory of SRAM-based FPGA systems. All the tools are intended to support designers in the assessment of the sensitivity to SEUs and in the generation of test patterns as early as possible during the design process and without requiring hardware prototypes of the system.

We point out that we do not propose these software tools as an alternative to radiation testing or fault injection, that should anyway be performed at the end of the design process on the final prototype of the system. We rather believe that such tools could be incorporated into the standard design process of FPGA-based systems, in order to allow designers to perform an early evaluation of the weaknesses of the system. In this way designers could get to the final fault injection or radiation test campaigns with a prototype of the system that has already been well studied and hardened and that because of this has great chance to successfully pass these tests, thus saving time and money.

1.3 Thesis Organization

This dissertation is organized as follows:

- **Chapter 2** introduces the basic concepts of the FPGA technology, it describes the generic architecture of an FPGA device in terms of logic, routing and I/O structure, it discusses pros and cons of the three FPGA programming technologies and finally it presents examples of applications of FPGA devices in various application domains;
- **Chapter 3** briefly introduces the natural radiation environment, and then it discusses the effects of radiation on digital devices with particular emphasis on SRAM-based FPGA devices and on the effects of SEUs occurring in the configuration memory of such systems;
- **Chapter 4** discusses the state of the art in the field of SEU sensitivity analysis; then it briefly presents the SAN formalism and Möbius environment that have been used to implement the presented SEU simulator and finally it presents the structure of the simulator, the SAN models the compose it, the main configurable parameters of the simulator and the steps that have to be performed to run it;
- **Chapter 5** first discusses the state of the art in the field of SEU untestability analysis and it then presents the SAL modeling environment and language and the LTL logic, and finally discusses the UA^{2}TPG SEU untestability analyzer and automatic test pattern generator, showing how to use the SAL specification language to model FPGA netlists and the LTL logic to specify untestability theorems, and then presenting the structure and the usage of the tool;
- **Chapter 6** first presents the state of the art in the field of automatic test pattern generation for digital circuits in general and for FPGA-based systems in particular, then it briefly introduces the basic concepts of genetic algorithms, and finally it presents the GABES environment for the generation of test patterns, showing its structure and presenting its parameters and usage;
- **Chapter 7** discusses the analysis environment in which all the presented tools work, showing the steps required by the execution of the tools, presenting the EDIF parser and the *E^{2}STAR* tool and the information that these two tools exchange with the SEU analysis and test environment;

- **Chapter 8** first presents the circuits used for the evaluation of the proposed tools and then reports results from their application;
- **Chapter 9** concludes the thesis.

2

The FPGA Technology

Field Programmable Gate Arrays (FPGAs) are pre-fabricated, electrically programmable, silicon devices, composed of programmable logic blocks, a programmable routing structure and programmable Input/Output pads. Since the birth of the integrated circuit technology in 1960s, many attempts have been done to achieve programmable devices in order to give hardware architects the chance of exploiting hardware performance and software flexibility at the same time.

The first FPGA was introduced by Xilinx in 1984 with the XC2064 logic cell array; since then the FPGA technology has grown in terms of scale of integration and performance. The ability of being programmed (and most of the times re-programmed) provides many advantages over other hardware technologies. Micro Controllers (μCs) and General Purpose Processors offer greater flexibility thanks to their programmability, but generally have much lower performance in terms of computation time and power consumption. In [183] and [184] Underwood *et al.* analysed performance differences between a general purpose CPU and an ad-hoc programmed FPGA in calculating vector dot product, matrix by vector multiplication and floating point addition, multiplication and division. In particular the number of MFLOPS that each technology is able to perform was considered as a quality factor. In these papers a trend analysis was also done, considering the growth rate of the integration scale in CMOS technology, and thus the growth rate of the number of common logic blocks in an FPGA and of the number of transistors in a CPU. The analysis was made between 1997 and 2009 and shows how the FPGA technology is generally able to reach 10^6 MFLOPS while CPUs still remain around 10^5 MFLOPS and in some cases, like floating point division, even at 10^3 MFLOPS while FPGAs already reach 10^5 MFLOPS.

Application Specific Integrated Circuits (ASICs) offer better performance than FPGA devices in terms of computational time, area requirements and power consumption. In [114] Kuon and Rose analysed performance gaps between ASIC and FPGA technologies. In their experiments 90nm CMOS SRAM FPGA and 90nm CMOS standard cell technology are considered. In that paper, area, delay and power gaps were estimated.

Area gap: For the ASIC technology the area occupation was considered to be the final silicon area used by the place&route process. For the FPGA technology only the area of the actually used logic blocks was considered. The analysis showed that FPGAs using only soft logic blocks are 35 times larger than ASICs on average; if also hard logic blocks are used, the average gap decreases to 18.

Time gap: For timing analysis the critical path of both ASIC and FPGA designs was considered. The analysis showed that FPGAs using only soft logic blocks are 3.4 times slower than ASICs on average; if the FPGAs included hard logic blocks the gap is 3 on average.

Dynamic Power Consumption gap: both for ASICs and FPGAs dynamic power consumption was calculated operating the device at the highest possible speed. The result was that FPGAs with only soft logic blocks have a dynamic dissipation 14 times higher than ASICs, while the gap decreases to 7,1 if hard logic blocks are used in FPGA architectures.

Kuon and Rose experimentally demonstrated how FPGA performance is still lower than ASIC performance; additionally they showed that the intensive use of hard logic blocks into FPGA architectures can significantly reduce the gap between the two technologies.

On the other hand ASICs require a much longer time to market and bigger economic effort: a full custom ASIC design needs a very long time and a high cost to be completed since state-of-the-art tools for synthesis, placement-and-routing, extraction, simulation, timing and power analysis, great engineering effort and very expensive foundry masks are needed. An FPGA design needs only between a few dollars and a few thousand dollars per unit, much less engineering effort and much shorter time to be designed and configured, and often an FPGA device can be reconfigured if a mistake was made during the design cycle.

Given this, only large scale productions can afford a full custom ASIC design, while small and medium scale productions prefer saving money and time by the use of FPGAs devices. For this kind of productions FPGAs represent nowadays the best trade-off between performance on one hand and cost and time to market on the other hand. Nowadays FPGAs have become the dominant programmable logic technology, no longer being used merely as glue logic or as prototyping devices and starting to be used to implement sub-systems or complete systems (what is called System-on-Chip). Leaders of the market of FPGAs are Xilinx, Altera, Microsemi, Atmel and Lattice.

2.1 FPGA Architecture

An FPGA is a prefabricated array of programmable blocks, interconnected by a programmable routing architecture and surrounded by programmable input/output blocks [115]. Figure 2.1 shows the basic architecture of an FPGA chip.

FPGA programming consists in defining the hardware structure of the system by producing a programming code, called a bitstream, that, after being downloaded into the configuration memory of the device itself, specifies the functionality implemented by LUTs and enables or disables a connections between wires to connect or disconnect two logic blocks or a logic block and an I/O pad.

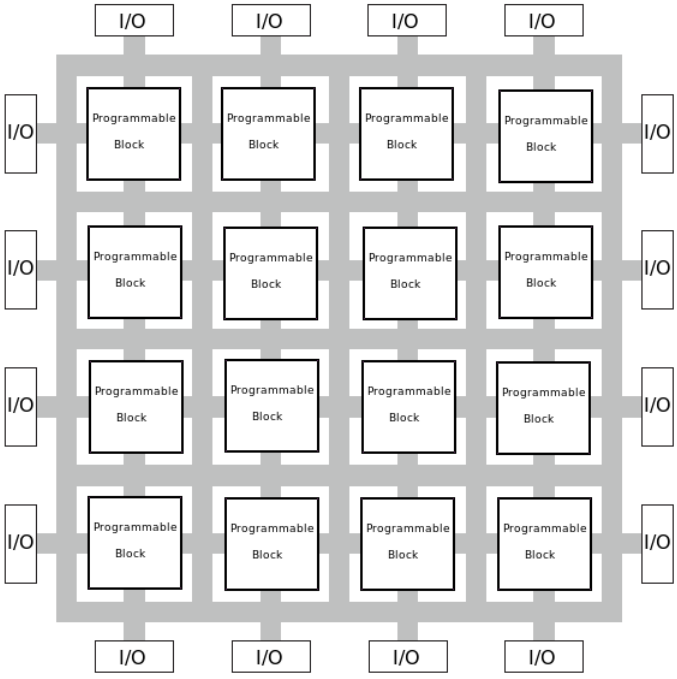


Figure 2.1. Basic FPGA structure [115].

Programmable blocks may be simple combinatorial logic (these blocks are called *Soft Logic Blocks*) or memories, multiplexers, ALUs and other kinds of prefabricated circuitry (these blocks are called *Hard Logic Blocks*).

Blocks may be programmed to implement a certain functionality, routing architectures may be programmed to interconnect various blocks and I/O pads may be programmed to provide off-chip connections.

In the remainder of this section we will first discuss common logic block features; we will analyse general features of the routing architectures and finally we will briefly discuss general characteristics of the I/O structure.

2.1.1 Logic Block Architecture

As said previously, an FPGA is an array of programmable soft blocks (simple programmable combinatorial logic) and hard blocks (memories, multiplexers, ALUs and

other prefabricated circuitry). The purpose of these blocks is to provide the basic computational and storage elements for the construction of the global logic system. The simplest logic block may be made of just one transistor; on the other hand some FPGAs have an entire processor as a basic logic block. The fine grain approach offers the maximum flexibility and programmability to the designer at the cost of a great area inefficiency due to the need of a large amount of programmable interconnections, low performance, because each routing "hop" is slow, and high power consumption. The coarse grain CPU-based approach offers a low level of flexibility to the chip designer and suffers of great inefficiency in implementing low level logic functions.

Usually, the basic programmable blocks of an FPGA are lookup tables (LUTs). A LUT can be represented as a small n-input memory, whose contents, stored in configuration bits, and specified by the bitstream, represent the output of the LUT itself. An n-input LUT can be used to implement any n-argument logic function. The logic structure of a 2-input LUT implementing the OR function is shown in Figure 2.2 (note that this is just a didactic example, while modern FPGA devices are equipped with larger 4-, 5- and 6-input LUTs), while the physical structure of the same 2-input LUT using SRAM a configuration memory is shown in Figure 2.3 (where the four boxes on the left-side of the figure represent the four configuration bits associated with the LUT).

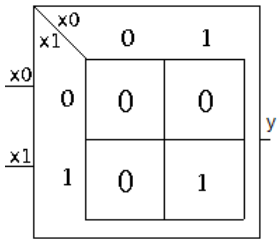


Figure 2.2. Logic structure of a 2-input LUT implementing the OR function.

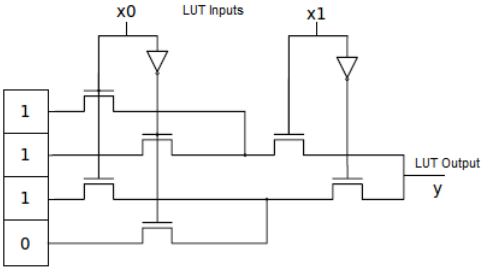
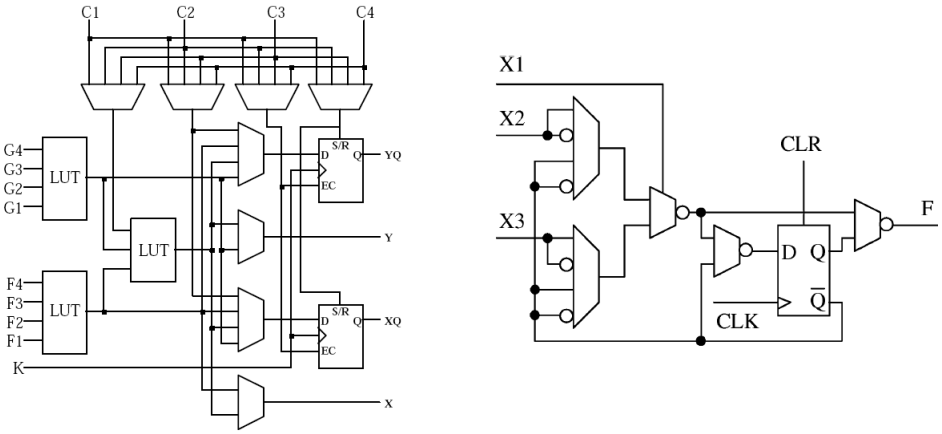


Figure 2.3. Physical structure of a 2-input LUT with static memory cells [115].

More complex configurable logic blocks (CLBs) may have many more input and output signals, more than one lookup table, some flip-flops and many multiplexers to select which input signals should drive the output. Figure 2.4 shows the structure of the basic logic block of a Xilinx 4000, that was produced in 1994, and of a Xilinx 6200, produced in 1996, as examples of commercial FPGA devices.



a) Xilinx 4000 Basic Logic Block

b) Xilinx 6200 Basic Logic Block

Figure 2.4. Basic logic block structure of a Xilinx 4000 and of a Xilinx 6200 [115].

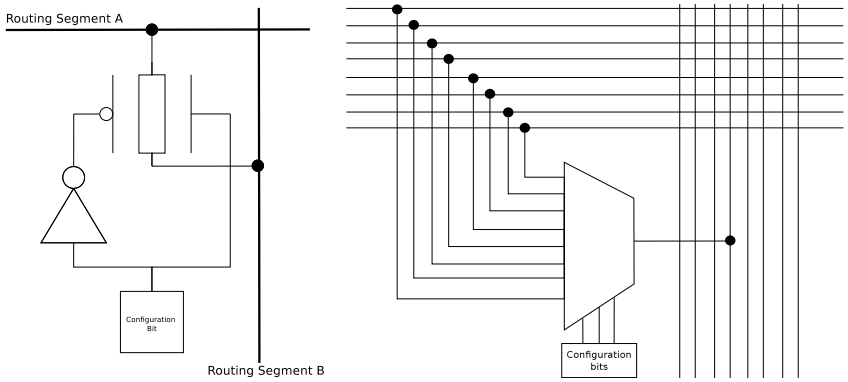
A Xilinx 4000 basic logic block has 12 inputs (C1 to C4, G1 to G4 and F1 to F4) and 4 outputs (Y, YQ, X and XQ), three lookup tables, eight multiplexers and two flip-flops. The Xilinx 4000 basic logic block can implement any two logical functions of four inputs (using the two 4-input lookup tables independently) or some logical functions of up to nine inputs (using all three lookup tables).

The basic logic block of a Xilinx 6200 has three input signals and only one output signal. It has two 4-input multiplexers, three 2-input multiplexers and a D-flip-flop. The output multiplexer is used to decide whether the output of the flip-flop drives the output of the logic block. Any basic logic blocks in a Xilinx 6200 can implement any two-input function and some three-input functions.

2.1.2 Programmable Routing Architecture

The programmable routing architecture in an FPGA provides connections among logic blocks and I/O blocks to compose a complete user-designed circuit. It consists of wires and switchboxes. According to [186], wires in an FPGA can be classified as follows:

- *segments*: connections between two switchboxes.
- *tracks*: sequences of one or more segments connecting two logic components.



(a) Pass transistor. (b) Multiplexer.

Figure 2.5. Structure of PIPs.

- *channels*: groups of parallel tracks.

Switchboxes are routing components that are configured to connect different wires. Switchboxes are programmed through *Programmable Interconnect Points (PIPs)*, that are configured by one of the programming technologies described in Section 2.2, that form the desired connections. The basic structure of a PIP is composed of a pass-transistor (see Figure 2.5(a)) or a multiplexer (see Figure 2.5(b)), that connects or disconnects two routing segments depending on the value of one or more configuration bits.

In order to give a high degree of flexibility and programmability to the designer, programmable routing architectures must offer fast and short wires to connect neighboring blocks, but also slower intermediate and long wires to connect more distant blocks. Another important issue is finding the right number of PIPs to grant a good degree of programmability without using too much silicon area and introducing too long delays. Two common switchbox architectures are currently employed in FPGAs (shown in Figure 2.6):

Disjoint [121], where wires connected to the four sides of the switchbox are grouped in four groups. In each group, each wire is univocally identified by an ID. Each wire can only be connected with wires belonging to different groups and having the same ID. This architecture imposes strict limitations to the programmability of the routing architecture, but it has a very low overhead in terms of silicon area employed for configuration bits associated with the routing structure.

Wilton [189], that removes the Disjoint switchbox restriction, allowing to a wire to be connected with wires with different IDs belonging to different groups. This architecture offers designers the highest flexibility at a high cost in terms of silicon area.

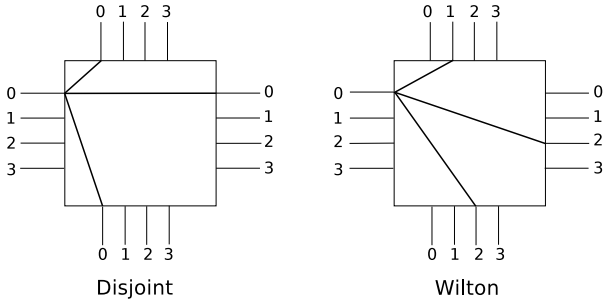


Figure 2.6. Disjoint and Wilton Switchboxes examples.

Two different programmable routing architectures for FPGA devices have been proposed in the literature, hierarchical and island-style. In the remainder of this subsection we describe these two routing architectures and discuss the respective advantages and drawbacks.

Island-Style Programmable Routing Architecture

In Island-style FPGAs, logic blocks are arranged in a two-dimensional mesh with routing resources evenly distributed throughout the mesh. An island-style routing architecture typically has routing channels on all four sides of the logic blocks. Logic blocks are grouped in *tiles*. A tile is defined as a number of CLBs connected together. Island-style routing architectures generally employ wire segments of different lengths in each channel in an attempt to provide the most appropriate length for each given connection.

This routing structure offers a number of advantages. Since routing wires of different lengths are in close physical proximity to logic blocks, a logic block can be efficiently connected to other logic blocks at different distances. As a result of this regularity, the minimum feasible routing delay between logic blocks can quickly be estimated. Figure 2.7 shows the common structure of an island-style programmable routing architecture.

It must be emphasized that the greater the number of programmable switches built in the FPGA, the greater is the area occupation and the lower the performance of the device. A great number of switches in fact gives a higher degree of flexibility and programmability to the designer, but produces a great consumption of silicon area and worse performance. Finding the best trade-off among these factors represents a still unsolved challenge for FPGA research.

The general structure of an island-style routing architecture can be divided in four levels, as discussed in [186], starting from a tile and arriving at the whole FPGA device:

- *Tile routing* (see Figure 2.8): composed of the hard-wired interconnection resources that connect the CLBs inside the tile.

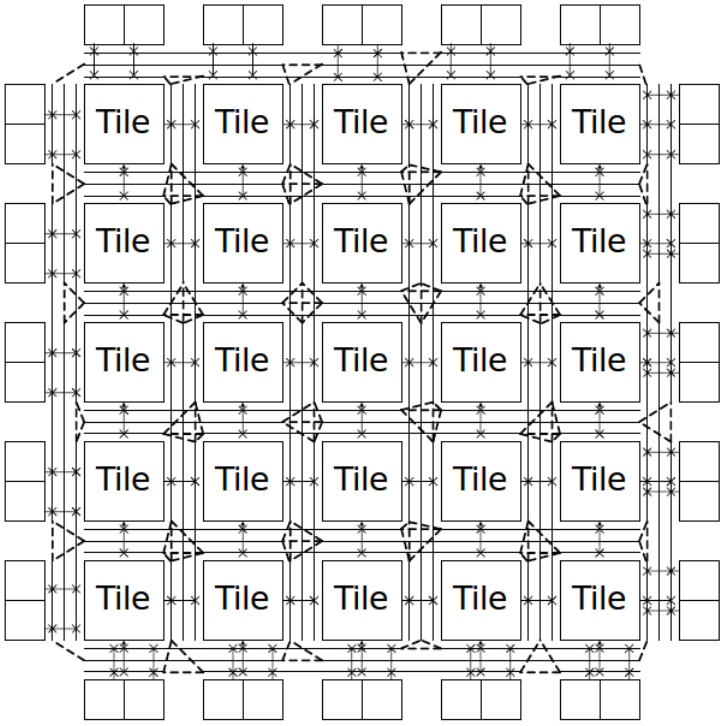


Figure 2.7. Island-style programmable routing architecture [115].

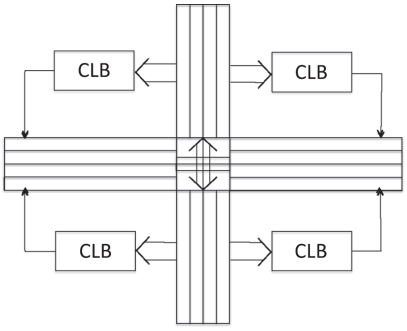


Figure 2.8. Tile routing [186].

- *Local routing* (see Figure 2.9): composed of wires that connect neighboring tiles. The interconnections between these wires are configured through programmable switchboxes.
- *Multiple-tile routing* (see Figure 2.10): composed of long wires with low resistance that can be traversed by signals in both the directions. These long wires are used

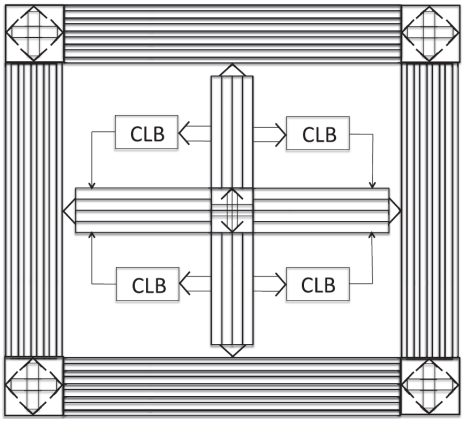


Figure 2.9. Local routing [186].

to connect distant tiles. These wires are connected through programmable switch-boxes.

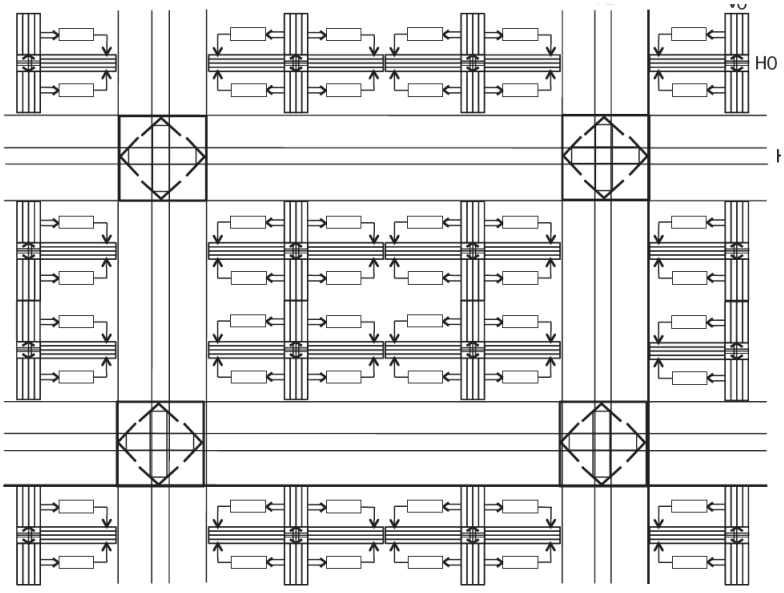


Figure 2.10. Multiple tile routing [186].

- *Context routing*: composed of the I/O pads that allow the system to be connected with the external environment.

Hierarchical Programmable Routing Architecture

A hierarchical programmable routing architecture consists of different buses crossing each other, interconnected through switchboxes positioned at the crossing points. In a hierarchical routing architecture logic blocks are separated into distinct groups. Connections between logic blocks within a group can be made using wire segments at the lowest level of the routing hierarchy. Connections between logic blocks in hierarchically distant groups require the traversal of one or more levels of the hierarchy of routing segments and so the traversal of one or more programmable switchboxes. Figure 2.11 shows the structure of a hierarchical programmable routing architecture.

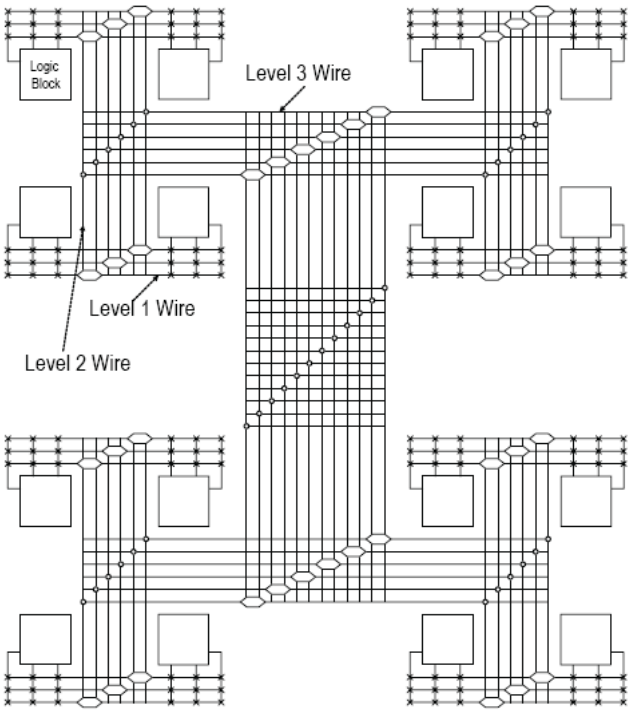


Figure 2.11. Hierarchical programmable routing architecture [115].

The only advantage of using hierarchical programmable routing architectures is that there is a quite predictable inter-block delay due to the regularity of the structure. Drawbacks are that jumping from a hierarchical level to another imposes a great delay, even if two logic blocks are physically close. In addition, even if, as a first approximation, intra level delays may appear constant, physical distances and differences in capacitance and resistance may produce a wide variation in inter-block delay.

For these reasons, commercial FPGAs do not use this type of global routing architecture and, instead, use only one level of hierarchy to create a flat, island-style global routing architecture.

2.1.3 Input/Output Architecture

As previously described, an FPGA device communicates with external components thruogh an input/output architecture composed of I/O pads disposed all around the FPGA's structure.

I/O pads influence both the rate at which the device will communicate with other devices and the total silicon area occupied, since they consume a significant part of the FPGA silicon area.

A crucial aspect of the FPGA architecture is the selection of which and how many interface standards the device might support through its I/O pads since, while a basic logic block is designed to implement almost any logic function, I/O pads are generally designed to implement only a small number of protocols (or even only one) protocol. Giving the designer the chance to choose which protocol should be implemented by each I/O cell would lead to a significant increase of silicon area occupation. On the other hand, giving this chance to the designer would increase the flexibility of the architecture.

An other challenge in input/output architecture design is the great diversity in input/output standards. For example, different standards may require different input voltage thresholds and output voltage levels. To support these differences, different I/O supply voltages are often needed for each standard.

2.2 Programming Technologies

Different technologies are used to store the programming code in an FPGA device. In the following we focus primarily on static memory programming technology, and then on the other two commonly used technologies: non-volatile memory (Flash and EEPROM) and anti-fuse.

2.2.1 Static Memory Programming Technology

In an SRAM-based FPGA, static memory cells, whose structure is shown in Figure 2.12, are distributed throughout the device to provide configurability. Static memory cells have two usages: store the functionality implemented by LUTs, and store PIP configurations in order to set interconnections among logic components.

Static memory has become the dominant FPGA programming technology thanks to two primary advantages: (i) reprogrammability, which means that a SRAM-based FPGA can be reprogrammed an indefinite number of times, and (ii) the use of standard CMOS process technology, which means that static memory FPGAs can benefit from

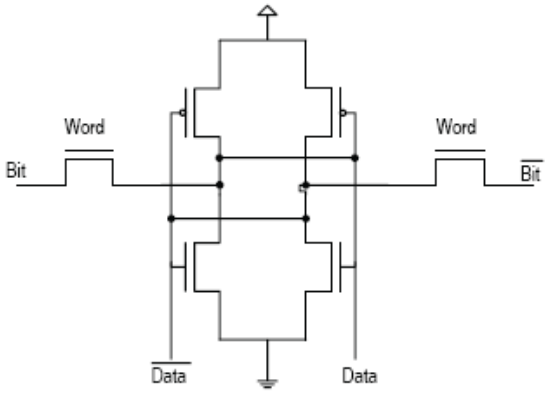


Figure 2.12. Static Memory Cell structure [115].

the higher speed and the lower dynamic power consumption of new CMOS processes with smaller minimum geometries.

Static memory programming technology has however the following drawbacks: (i) Large size, because static memory programming needs a 1-bit memory cell for each programmable gate and connection; (ii) volatility, because static memory needs a power supply to store data, so it needs a non-volatile memory technology to store configuration data when the device is powered down and from which the device reads it during start up, so increasing the total silicon area occupied by the device, and (iii) lower security because, since configuration information must be loaded from the non-volatile memory to the static memory at power up, there is the possibility that this information could be intercepted and stolen for use in competing systems.

Static memory cells are used as basic storage units in the Xilinx Virtex [198, 199, 200], and Spartan [197] device families, in the Altera Stratix [10, 9, 12, 13, 16, 17, 22], Arria [18, 19] and Cyclone [14, 11, 15, 20, 21] device families and in the Lattice ECP3 [119] and ECP2 [118] devices.

2.2.2 Flash/EEPROM Programming Technology

An alternative to static memory is the use of Flash Memory and EEPROM. The greatest advantage of these technologies for FPGA programming is non volatility: using this kind of memories it is no longer necessary to have an external resource to store and load configuration data when the device is powered down. Additionally, a flash or EEPROM based device can be used immediately after power-up, since configuration data are stored in the configuration memory itself.

Drawbacks of the use of these technologies for FPGA configuration are primarily that Flash and EEPROM memory do not use the standard CMOS process technology, so they are generally quite slower and larger than static memory, and that they

cannot be reprogrammed an indefinite number of times, like static memory: generally a Flash or EEPROM chip can be reprogrammed up to 500 times, which however may be sufficient for most of the uses of FPGAs. It has to be noticed that also using Flash/EEPROM as an FPGA programming technology, a 1-bit memory cell is required for each programmable gate or connection, so also Flash/EEPROM based programming approach suffers from great silicon area occupation.

Flash Memory or EEPROM are used as basic storage units in the Microsemi ProASIC3 [136, 137, 138, 141] and Igloo [135] device families, in the Lattice XP [117] device family and in the Atmel AT40K [25] family.

2.2.3 Anti-Fuse Programming Technology

An anti-fuse device is a structure that exhibits very high resistance (like an open circuit) under normal circumstances and can be programmed by applying a high voltage to the gate creating a low resistance link that becomes a connection.

The structure of an anti-fuse device is shown in Figure 2.13.

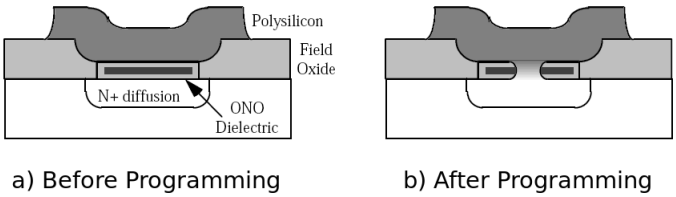


Figure 2.13. Anti-fuse structure [98].

Two anti-fuse technologies exist; the first, called *Dielectric anti-fuse*, uses an oxide-nitride-oxide dielectric between the channel and the gate; by applying a high voltage the dielectric breaks down and forms a conductive link. This approach has been largely replaced by the *metal-to-metal-based anti-fuse* technology: this approach uses an insulating material, such as amorphous silicon or silicon oxide, to insulate two metal layers; when applying a high voltage to the gate, the insulating layer breaks down and leaves an interconnection between the two metal layers.

The great advantage of using anti-fuse as an FPGA programming technology is that, as the anti-fuse is inside the transistor itself, no more silicon area is required for programming than the area needed by the logic structure. Other advantages are non-volatility and low resistance, and thus, low power consumption.

Drawbacks of using anti-fuse for FPGA programming are that anti-fuse devices need a non standard CMOS process technology and that an anti-fuse device can be programmed only once, so anti-fuse based FPGAs need a more accurate phase of simulation before programming, since bug corrections during the test phase need the use of a new, blank chip.

Anti-fuse is used as basic storage units in the Microsemi RTAX [140, 139], SX-A [134], eX [132] and MX [133] device families.

2.3 FPGA Application Fields

When FPGAs were first introduced they were primarily considered to be just another form of gate array. Although they had lower speed and capacity, and had a higher unit cost, they did not have the large startup costs and long design times necessary for gate array programmable devices. Thus, they usually were used for implementing random logic and glue logic in low volume systems with limited speed and computational power demands. If the computational power of a single FPGA was not enough to handle the desired computation, multiple FPGAs could be included on the same board, distributing the computation among these chips. However FPGAs are more than just slow and small gate array devices.

The distinguishing feature of (SRAM-based) FPGAs is their in-circuit reprogrammability. Since their programming can be changed quickly, without any rewiring or refabrication, they can be used in a much more flexible manner than standard gate arrays. An example of this is multi-mode hardware: hardware systems in which two functions have to be performed in a mutually exclusive manner; using ASICs, two different chips are needed, while using the FPGA technology the same device, with two ROM chips with the configurations implementing the different functions is sufficient; when the device has to stop performing a function and start performing the other, it has just to read the new configuration from the right ROM chip and reconfigure itself.

Nowadays FPGA performance, although not comparable with ASIC performance neither in terms of computational power nor in terms of silicon area occupation, allows the FPGA technology to be deployed in almost any application field, both safety critical and not safety critical, also because while the ASIC design and production costs and time to market have grown dramatically, FPGA devices are undoubtedly cheap and have a quite short time to market. In fact the FPGA technology is nowadays the leading technology for all those applications that need good performance and that have strong time to market requirements or that require a small scale production.

2.3.1 Hardware Prototyping

Prototyping and emulation of hardware devices of other, more expensive, technologies, has been one of the first uses of FPGAs. The basic idea is that the designers of a custom ASIC or general purpose processor need to make sure that the circuit they designed correctly implements the desired computation. Software simulation can perform these checks, but does it quite slowly. In logic emulation, the circuit to be tested is instead mapped onto an FPGA or a multi-FPGA system, yielding results several orders of magnitude faster than software simulations.

Examples of processor prototyping using FPGA devices are: [192] where a Xilinx Virtex II is used to implement a fully programmable prototype of an Intel Itanium processor; [92] where a Xilinx XC4000 is used to prototype a MIPS RISC processor; [154] where a Xilinx XCV2000E is used to implement a prototype of a 4-way superscalar speculative out-of-order processor executing a bubble sort on 100 random integers.

2.3.2 Aerospace and Defense

A field where FPGAs find many applications is Aerospace and Defense due to the enormous number of tasks performed by digital devices, historically mainly ASICs.

FPGA devices are designed for Firing Control and Aiming Control, Hyperspectral Vision, Sonar, Radar and Radio Systems, Missile Launching Platform Control, Flight Control and Crew Assisted Operations, System Fault Tolerance, Control of Unmanned Aerial Vehicles (*UAV*), Unmanned Ground Vehicles (*UGV*) and Autonomous Underwater Vehicles (*AUV*).

Examples of Aerospace and Defense FPGA applications are: [38] where a Xilinx Spartan 3 device, together with two AMD Geode NX1500, is used to implement a Fuzzy Logic based Autonomous Motorcycle Platform; in [90] a Xilinx XC4000XL device is used instead of a Digital Signal Processor to implement a Sonar system; [120] where Xilinx XC2S100 devices are used to implement motes in a sensor network used to detect the direction from which a sniper is shooting in an urban battlefield; in [62] four ad hoc designed FPGAs, together with two Digital Signal Processors, two Digital to Analog Converters, two Analog to Digital Converters, and an on-board RAM, are used to implement the ASPECT board which is a general-purpose computing platform suited for communications-related signal processing, used to implement a robust, high-speed frequency-hopped battlefield radio; in [88] Peterson and Drager discuss how the adoption of FPGA devices could improve and accelerate military applications such as hyperspectral imaging or chemical reactions simulations; in [48] an ad hoc designed Flash memory based FPGA, together with a Digital Signal Processor and an FSC20 CMOS image sensor, is used to design and implement an autonomous flight control system for the GTMax Research Unmanned helicopter and for the HeliSpy 11-inch ducted fan UAV; in [42] three Xilinx Spartan 3 are used to implement a PC/104-Plus, which is the base board for a Urban/Indoor Network-Assisted GPS-based Navigation System.

2.3.3 Automotive

FPGAs are also widely employed in automotive applications. FPGA devices are used in a great number of tasks, among which: Rear Seat Entertainment, Driver Assistance Systems, Adaptive Cruise Control, Lane Departure Warning, Park Assistance,

Back Guide Monitor, Drowsy Driver Detection, Head-up Display, Night Vision, Window Wiper Control, GPS, Diagnostics, Engine Management, Steering and Braking Assistance.

In [103] a survey of possible FPGA applications in the automotive field is done; in [97] an Altera Cyclone II is used to implement a car radar system while in [129] the same device is used to design a rear seat entertainment system.

Many details about FPGA usage in the automotive application field can also be found in commercial brochures by Xilinx ([113] and [194]), Altera [58] and Lattice [59].

2.3.4 Cryptography and Network Security

As FPGAs represent a very good technology base for those applications that are simply performed in hardware, that do not need high performance and cannot afford large ASIC design and production costs, a great number of cryptographic tasks can be performed by FPGA based systems or even by a single ad hoc designed FPGA device. There is a great number of examples of cryptographic algorithms implemented in an FPGA: in [150] a Xilinx Virtex XCV1000BG560-4 FPGA is used to implement and evaluate the AES algorithm; in [68] the same authors use the same device to implement and evaluate the Serpent block cipher; in [91] Grembowski *et al.* uses the same FPGA to implement and comparatively analyse the SHA-1 and SHA-512 hash functions; in [166] Runje and Kovac develop an ad hoc FPGA architecture called UNICORN to implement the IDEA algorithm; in [124] an Altera Flex 10KE device is used to implement the BlowFish encryption algorithm; in [67] Kaps and Paar implement the DES encryption algorithm in a Xilinx low power device and then compare the obtained performance with those obtained by an ASIC implementation of the same algorithm; in [128] Mazzeo *et al.* implement the RSA public key encryption algorithm in a Xilinx Virtex-E 2000-8 and then compare the obtained performance with a previous implementation in a Xilinx XC40250XV-09; in [64] a Virtex V1000FG680-6 is used to implement the md5 hash function; in [87] Gosset, Standaert and Quisquater implement the SQUASH hash function in a Xilinx Virtex 4 LX FPGA; in [158] a Xilinx XC40200XV-9-BG560 device is used to implement the RC-6 and the CAST-256 encryption algorithms; finally in [126] a Xilinx Virtex II Pro is used to implement the IPSec Internet security protocol.

2.3.5 Railways

The FPGA technology is emerging also in the railway application field, both for safety critical and non safety critical functions, for example in [65] Dobias and Kubatova describe the design of a railways interlocking system based on the FPGA technology while in [167] De Ruvo *et al.* describe an FPGA based design of an automatic hexagonal bolts detection system for the railway maintenance developed in an Altera Stratix device.

2.3.6 Digital Signal Processing, Industrial and Nuclear Power Plant Control and Medical Applications

Another field where FPGAs have started being deployed thanks to their ever increasing computational power is Digital Signal Processing in substitution of classical Digital Signal Processors, which are custom, high-performance, high-cost devices.

FPGAs are used for instance for Fourier and Fast Fourier Transform as in [96] where an Altera Apex is used or in [123] where an Altera Stratix is used, for Multiply and Accumulate, for signal filtering as in [47] where an Altera Stratix is used to implement a Kalman Filter or [86] where a Xilinx XC6200 is used to implement a Viterbi decoder. FPGA devices are also commonly used for audio, image and speech processing. A survey about FPGA usage in Digital Signal Processing is [181] by Tessier and Burleson.

Closely related with Digital Signal Processing is the field of Industrial Control applications, where the FPGA technology is nowadays emerging. In [143] Monmasson and Cirstea present a survey of possible FPGA applications in industrial control systems.

Another application field tied to Digital Signal Processing is the Medical one: FPGA begin to be used for image diagnostics, as in [122] where Li *et al.* use a Xilinx Virtex II Pro to implement a platform for real-time 3D cone-beam CT reconstruction or [50] where a Xilinx Virtex 1000 is used to implement a parallel-beam backprojection system, but also for surgical and laboratory high-tech tools, and cardiac devices. A survey on FPGA applications in the medical field is presented by Goddard and Trepanier in [83].

As proved by recent papers as [23], [26] and [39], that discuss guidelines and proposals for the design, verification and validation methodologies of FPGA based instrumentation and control systems for nuclear power plants, the FPGA technology is also gaining interest in the nuclear control application field. As an example we cite [173], where the programmable digital comparator of the shutdown system no.1 of the CANadian Deuterium and Uranium (CANDU) reactor is implemented in an FPGA device.

2.3.7 Broadcast, Wired and Wireless Communication

A great number of communication protocols can be implemented using the FPGA technology: Audio-Video Bridging (*AVB*), High Definition-Serial Digital Interface (*HD-SDI*), Digital Video Broadcasting (*DVB*), MPEG and JPEG compressors, Code Division Multiple Access (*CDMA*), Orthogonal Frequency Division Multiple Access (*OFDMA*), WiMax, WiFi, ATM, TCP, Ethernet and SONET are only some examples.

Many other details about FPGA usage in the communication field can be found in commercial brochures by Xilinx ([193] and [196]) and Lattice ([60] and [61]).

3

Effects of Radiations on SRAM-based FPGAs

3.1 The Natural Space Radiation

The near-Earth natural radiations can be divided in two categories, the particles trapped in the Van Allen belts and the transient radiation. The particles trapped in the Van Allen belts are composed of protons, electrons, and heavy ions. The transient radiation consists of galactic cosmic ray particles and particles from solar events (coronal mass ejections and flares). The cosmic rays include all ions of all elements in the periodic table. The solar eruptions produce protons, alpha particles, heavy ions, and electrons [145].

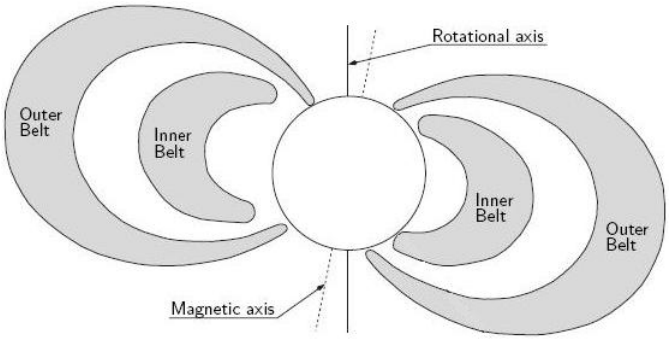
3.1.1 Trapped Protons and Electrons

The radiation belts consist principally of electrons and protons. These particles are trapped in the Earth's magnetic field. Their motions in the field consist of a gyration about field lines, a bouncing motion between the magnetic mirrors found near the Earth's poles, and a drift motion around the Earth [71] (see Figure 3.1).

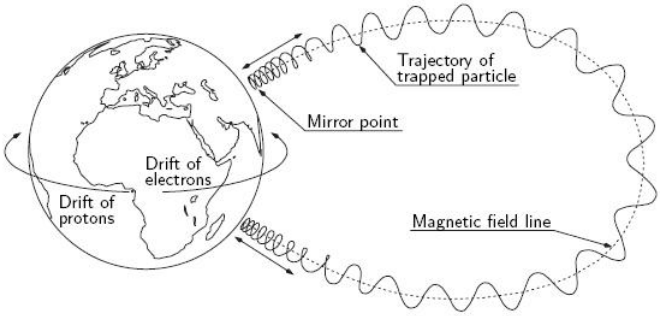
The trapped particles represent a significant threat for electronic systems. There are large variations in the level of hazard depending on the orbit of spacecraft and on the solar activity. Protons are especially problematic because of their high energies and penetrating power. Low energy electrons are the cause of electrostatic discharging which can be a serious problem for spacecraft in higher altitude orbits (e.g., geostationary) where they are exposed to intense fluxes of electrons. Higher energy electrons can penetrate into a spacecraft, collect in insulator materials, and discharge causing damage to electronics [145].

3.1.2 Galactic Cosmic Ray Heavy Ions

Cosmic Rays originate outside the solar system. Fluxes of these particles are low but, because they include heavy ions of elements such as iron, they cause intense



(a) Earth radiation belts.



(b) Basic motion of trapped particles in the earth magnetic field.

Figure 3.1. Trapped Particle Belts [71].

ionisation as they pass through materials. It is difficult to shield against these type of ions, and therefore they constitute a significant hazard [71].

Moreover, they have a high rate of energy deposition as measured by their *linear energy transfer (LET)* rate. A particle's LET is primarily dependent on the density of the target material and, to a lesser degree, the density and thickness of the shielding material [145].

3.1.3 Solar Particles

During solar events, large fluxes of protons are produced which reach Earth. Such events are unpredictable in their time of occurrence, magnitude, duration or composition. The Earth's magnetic field shields a region of near-Earth space from these particles (geomagnetic shielding) but they easily reach polar regions and high altitudes such as the geostationary orbit [71] (see Figure 3.2).

The particles from solar events are a concern for spacecraft designers. For systems that must operate during a solar particle event, the effect that both the solar protons and the solar heavy-ions have on the system must be evaluated. It is especially

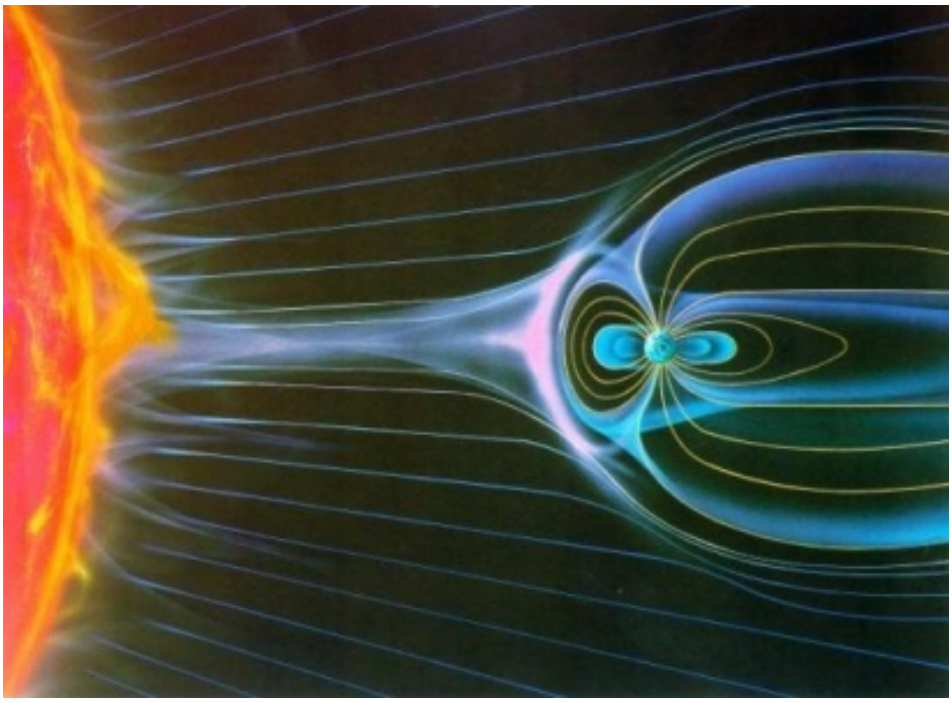


Figure 3.2. Geomagnetic Shielding [146].

important to take the peak flux levels into consideration. When setting requirements and operational guidelines for electronic devices, one must remember that peak solar particle conditions exist for only a small part of the total mission time [145].

3.2 Radiation Effects on Digital Devices

The effects of the natural space radiation on digital devices may be divided into two categories: long-term and short-term. An alternative classification is between ionizing and non-ionizing effects. Short-term ionizing effects are *single event effects (SEE)*. The long-term ionizing effect is the *total ionizing dose (TID)*. Finally the long-term non ionizing effect is the *displacement damage dose (DDD)*. TID and DDD effects consist of cumulative performance degradation of the device, that are generally visible after some time, and that almost uniformly affect the entire device. SEEs occur stochastically, at any time and they are generally visible after a very short time and only in the small region of the device affected by the particle strike [112, 144, 188].

3.2.1 Total Ionizing Dose

TID is a long-term degradation of electronic components due to the cumulative charge deposited in the silicon. Figure 3.3 shows the normal operation of a CMOS transistor

and its faulty operation due to TID. In particular TID in CMOS devices is caused by the cumulative charge trapped in the oxide layers of transistors. The effect of TID in digital devices is a progressive shift of the threshold voltage. Thus a digital device affected by TID has an increasing power consumption, due to an increasing leakage current, and a decreasing timing performance due to the threshold voltage shift itself, that makes the activation of a gate slower and slower. Finally, when the threshold voltage shift is large enough TID causes a functional failure of the system due to the impossibility of the activation of the gate. Significant sources of TID include trapped electrons, trapped protons, and solar protons [112, 127, 144, 148, 188].

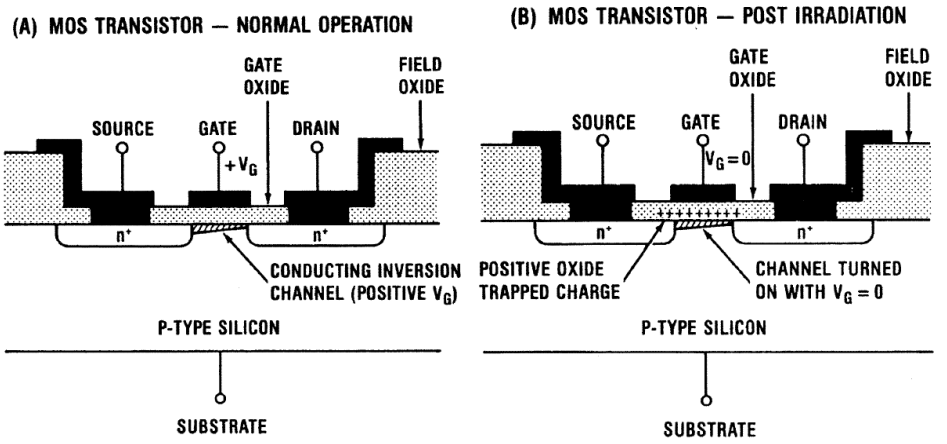


Figure 3.3. Normal (A) and post-irradiation (B) operation of a CMOS gate [148].

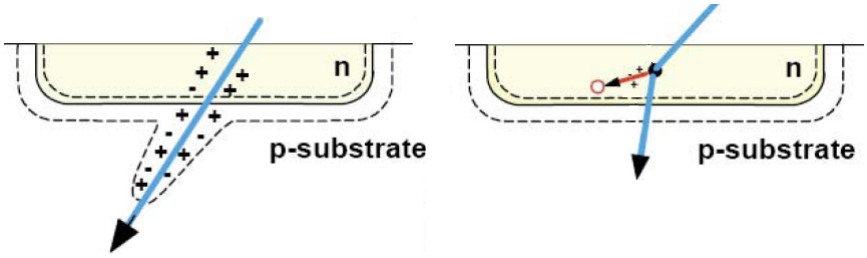
3.2.2 Displacement Damage Dose

DDD often has similar long-term degradation characteristics to TID, but it is a separate physical mechanism. DDD is essentially the cumulative degradation resulting from the displacement of nuclei in a material from their lattice position. Over time, sufficient displacement can occur and may change the device or material performance properties. Sources of DDD include trapped protons, solar protons, neutrons, and to a lesser extent for typical electronic systems, trapped electrons. It should be noted that technologies that are tolerant to TID are not necessarily tolerant to DDD and viceversa. In particular CMOS devices are immune to DDD, while BJT-based devices are deeply affected by DDD [144].

3.2.3 Single Event Effects

SEEs occur when a single ion strikes a material, depositing sufficient energy to cause an effect in the device. SEEs may be caused either through the ion's primary strike,

called direct ionization (see Figure 3.4(a)), or by the ion's secondary particles that issue from the strike, called indirect ionization via protons (see Figure 3.4(b)). The ionization induced by the particle strike induces a charge collection in the pn-junction of the transistor. An SEE is triggered and its type is defined according to the localisation and amount of collected charge and on the type and technology of the affected device [72]. The many types of SEE may be divided into two main categories: soft errors, that cause temporary malfunctions of the device, and hard errors, that cause permanent damages in the device structure [112, 144, 188].



(a) SEE induced by a heavy ion. (b) SEE induced by a proton.

Figure 3.4. Mechanism for Single Event Effects [72].

Soft errors occur when a transient pulse or bit-flip in the device causes a change in the content of a flip-flop or a memory element, or in a signal on a wire. Therefore, soft errors are entirely device and design specific, and are best categorized by their impact on the device. These are "soft" errors in the sense that a reset or rewriting of the device causes normal device behavior thereafter. Hard errors may be physically destructive for the device, and may cause permanent functional effects [112, 188, 144]. SEEs may be classified as follows [144]:

- Soft Errors
  - *Single Event Upset (SEU)*: a change of the content of a memory location or a flip-flop that causes a change of the state of the system.
  - *Single Event Transient (SET)*: a transient change of the value of the signal on a wire due to a current pulse.
- Hard Errors
  - *Single Hard Error (SHE)*: an SEE which causes a permanent change to the operation of a device. An example is a stuck bit in a memory device.
  - *Single Event Latchup (SEL)*: a condition which causes loss of device functionality due to a single event-induced high current. SELs are hard errors, and are potentially destructive since their effect is a high operating current. The latched condition can destroy the device or damage the power supply. An SEL is cleared by a power off-on reset or power strobing of the device. If power is not removed quickly, catastrophic failure may occur due to excessive heating.

SEL is strongly temperature dependent: the threshold for latchup decreases at high temperature.
- *Single Event Burnout (SEB)*: a condition which can cause device destruction due to a high current in a one or more transistors.
- *Single Event Gate Rupture (SEGR)*: a single ion-induced condition in transistors which may result in the formation of a conducting path in the gate oxide.

Single Event Upsets in SRAM

When an energetic particle strikes an SRAM cell (typically the drain of a transistor in the "off" state) the charge collected in the transistor's junction causes a transient current in the transistor. This current propagates through the feedback loop of the SRAM cell to the input of the other inverter of the SRAM cell. If the width and amplitude of the current pulse are sufficient, the next inverter will change its output and thus a new value will be loaded in the memory cell. The sensitivity of an SRAM cell depends on the gate capacitance and operating voltage. The gate capacitance together with the transistor channel resistance acts as a low pass filter that may reduce the rising slope and magnitude of the induced current pulse. With technology down-scaling the operational voltage of a device is also decreased. This means that less charge is needed to induce an SEU. [66, 188]. The basic mechanism of SEUs in SRAM cells is shown in Figure 3.5

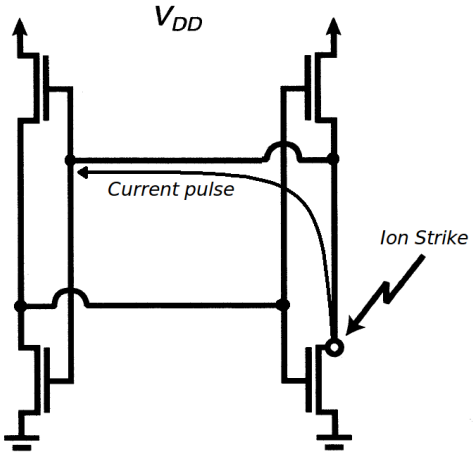


Figure 3.5. SEU mechanism in a SRAM cell [66].

3.3 Effects of Radiation on FPGA devices

The circuit elements affected by radiation in FPGA devices may be divided in two classes: user memory elements and configuration memory elements. The first class

(user memories) consists of the memory elements that can be used by the designer to implement an application, such as flip-flops, registers, and embedded memories. The second class (configuration memories) is further divided into memory cells that define the functions of logic blocks, and cells that establish their interconnections, as well as the registers and the flip-flops (FFs) of the several control mechanisms of the FPGA configuration itself.

User memories are affected by both TID and SEEs without respect of the configuration technology employed by the considered device. Configuration memories of anti-fuse-based FPGAs are completely immune to radiation. Flash/EEPROM-based FPGAs are immune to SEEs, but they are particularly prone to TID. Finally SRAM-based FPGAs are extremely prone to SEEs [188].

SEUs affecting the configuration memory represent the main cause of failure in SRAM-based FPGAs for two reasons: (i) the high occurrence rate of SEUs makes TID negligible and (ii) the impact of SEUs in the configuration memory is more relevant than faults in other resources, because the memory cells used for FPGA configuration are larger than those found in high-density devices used for storage [79].

3.3.1 Effects of Single Event Upset in the Configuration Memory of SRAM-based FPGA devices

An FPGA can contain millions of configuration bits controlling the routing structure and the logic blocks. SEUs in the configuration memory of an SRAM-based FPGA may disrupt the routing architecture of the implemented circuit and the behaviour of the functional units. Such faults may be considered permanent because the configuration memory is usually not written again after the first configuration.

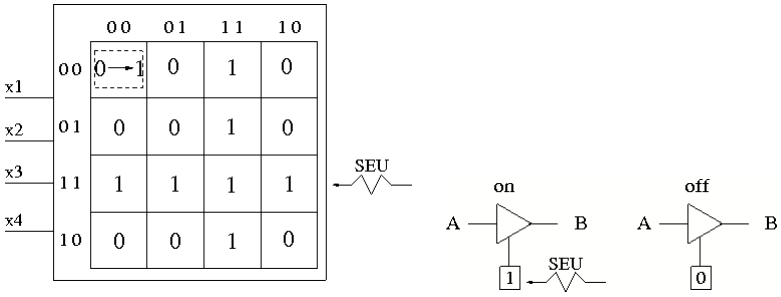
From a modeling point of view, SEUs affecting the configuration memory of an FPGA device may not be modeled by the stuck-at model that is generally assumed for digital circuits. A more accurate fault model has to be considered, as shown in [155] for logic resources and in [186] for routing resources. In the following we present the fault model that has been considered in the design of the presented software tools.

The model of SEUs affecting configuration bits controlling logical resources

The tools presented in this thesis adopt the functional fault model for SEUs affecting the configuration bits controlling the logic resources of FPGAs proposed in [155]. This fault model has been demonstrated to be much more accurate than the stuck-at fault model when the problem of analysing the effects of SEUs in SRAM-based FPGAs is addressed.

In the stuck-at fault model, a SEU in the configuration memory of a component causes the output of the component to be stuck at a given value, thus the faulty component always produces an incorrect value. In our simulator, a SEU in the configuration memory of a LUT causes an alteration of the functionality performed by the LUT. In particular, the faulty LUT will produce an incorrect value only when the configuration

of its input values is the one associated with the faulty configuration bit, while for every other configuration of its input values the faulty LUT will behave correctly. Figure 3.6(a) shows a SEU causing a bit flip in the configuration bit associated with input (0 0 0 0). In this case the logic function implemented by the LUT changes from the correct function $y = x_1 \cdot x_2 + x_3 \cdot x_4$ to the faulty function $y_f = x_1 \cdot x_2 + x_3 \cdot x_4 + \bar{x_1} \cdot \bar{x_2} \cdot \bar{x_3} \cdot \bar{x_4}$.



(a) SEU in a lookup table. (b) SEU in a I/O buffer.

Figure 3.6. Effects of SEUs in the logic components of an FPGA.

It may be observed that in the example the behaviours of y and y_f are different only when the values of the input signals are (0000).

A SEU in the configuration bit of an I/O buffer causes an undesired connection or disconnection between two wires, as shown in Figure 3.6(b).

The model of SEUs affecting configuration bits controlling routing resources

To show the model of SEUs [186], let us consider the switch block shown in Fig. 3.7, where two PIPs connect wire A to B and wire C to D, respectively. Depending on the position and the electrical properties of the affected PIP, an SEU in the routing structure can cause the following topological modifications (also shown in Fig. 3.7):

- *Open*, where the PIP is not programmed any more and thus the corresponding connection ($A \rightarrow B$) is deleted.
- *Antenna*, where a new connection ($\mathrm{unused} \rightarrow B$) is added between an unused input node and a used output node.
- *Conflict*, where a new connection ($A \rightarrow D$) is added between a used input node and a used output node.
- *Bridge*, where an existing connection is deleted ($C \rightarrow D$) and a new one ($A \rightarrow D$) is added between a used input node and the output node of the deleted connection.
- *Unrouted*, where the modification of the routing structure of the system induced by the SEU in the configuration bit controlling the PIP cannot be classified in any of the previous categories.

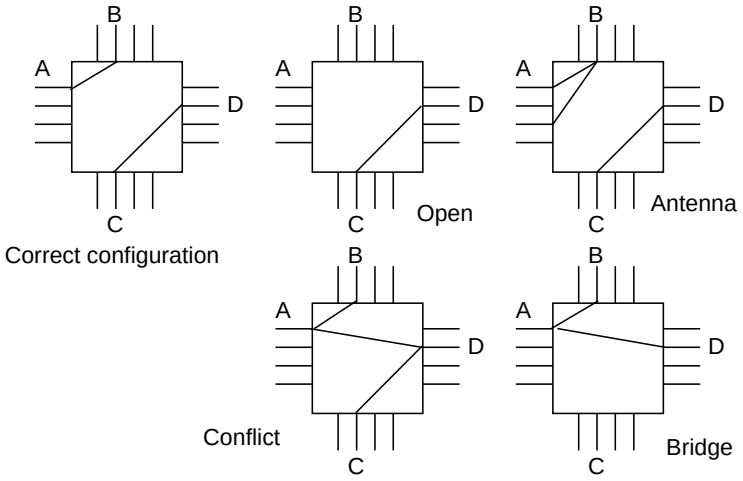


Figure 3.7. Effects of SEUs in the routing components of an FPGA.

Effects of SEUs in PIPs involving unused connections are not considered as they do not cause a faulty behaviour since they do not affect the system.

Figure 3.8 shows the distribution of the effects of SEUs in the configuration bits controlling routing resources [186]. It appears evident that the major effects are open and conflict.

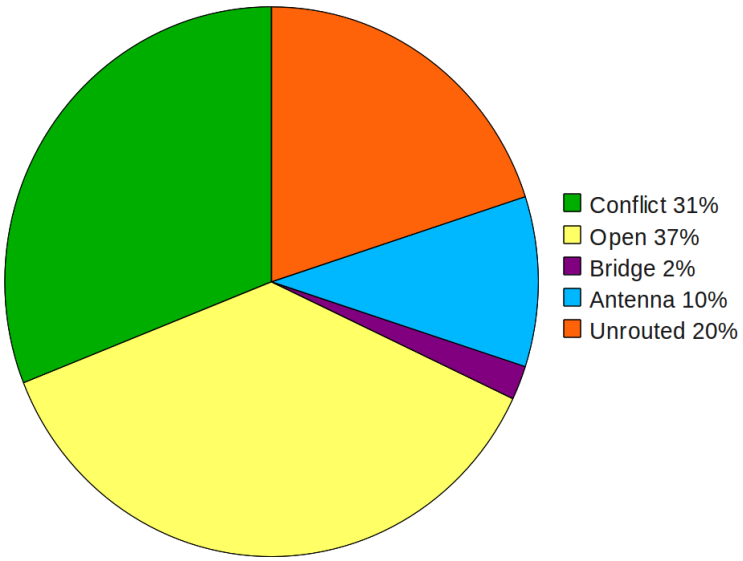


Figure 3.8. Distribution of the effects of SEUs in the configuration bits controlling routing resources [186].

The effects of the modification of the routing structure of the system induced by SEUs in configuration bits controlling PIPs can be mapped on modifications of the behaviour of logic components of the netlist at a higher level of abstraction. The following logical effects can be induced by SEUs in configuration bits controlling routing elements [186]:

- Stuck-at: A node is stuck at a constant logic value.
- Bridge: Two nodes exchange their values.
- Wired-AND (Wired-OR): The value of a node C is the AND (OR) of the values of two nodes A and B.
- Wired-MIX: The values of two nodes A and B are mixed as follows: If the values A and B are equal, A and B keep their correct values, otherwise A takes the zero logic value and B takes the one logic value.

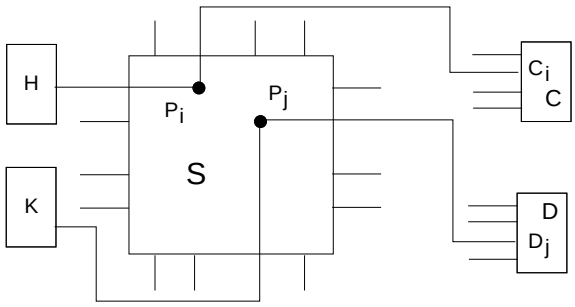


Figure 3.9. Routing example.

With reference to Fig. 3.9, if S is a switch box, C and D are two components directly connected to S, C_i and D_j are the input pins of C and D connected to S through the PIPs P_i and P_j, respectively, the five possible effects of a SEU in the configuration bit controlling P are modeled as follows:

- A stuck-at on P_i is modeled by setting the logic signal on C_i at the corresponding fixed value.
- A bridge between P_i and P_j is modeled by exchanging the logic values on C_i and D_j.
- A Wired-AND (Wired-OR) between P_i and P_j is modeled by setting the logic signals on C_i and on D_j to the value P_i AND (OR) P_j.
- A SEU causing a Wired-MIX between P_i and P_j is modeled by setting the logic signals on C_i to 1 and on D_j to 0 if $P_i \neq P_j$, while leaving C_i and D_j unaltered otherwise

It may be observed that a given SEU in the configuration bit associated with a PIP can propagate to different routing segments, and that the same SEU can have different effects on the routing segments through which it propagates.

Table 3.1. Correspondence between physical and logic effects of SEUs in tile routing PIPs.

Physical Modification	Logic Effect
Conflict	Wired-AND
Open	Stuck-at 0
Bridge	Bridge
Antenna	Stuck-at 0

Table 3.2. Correspondence between physical and logic effects of SEUs in tile routing PIPs.

Physical Modification	Logic Effect
Conflict	Wired-AND
	Wired-MiX
Open	Stuck-at 1
Bridge	Bridge
Antenna	Stuck-at 0

Table 3.3. Correspondence between physical and logic effects of SEUs in tile routing PIPs.

Physical Modification	Logic Effect
Conflict	Wired-AND
Open	Stuck-at 0
Bridge	Bridge
Antenna	Stuck-at 0

In particular the effect of a SEU in the configuration memory controlling a PIP depends on the position of the PIP with reference to the levels of routing introduced in Section 2.1.2. The logical effects corresponding to the physical modifications induced by SEUs in the configuration bits controlling PIPs belonging to the tile routing, local routing and multiple-tile routing are summarized in Table 3.1, Table 3.2 and Table 3.3 respectively [186].

4

The ASSESS Tool

4.1 Related Work: Techniques for SEU Sensitivity Analysis

A lot of techniques and tools for the analysis of the effects of the various types of faults in digital circuits have been proposed in the last years. A detailed discussion about this topic can be found in [201]. In the remainder of this section we focus on the techniques and tools specifically focused on the analysis of the effects of SEUs in SRAM-based FPGAs that can be found in the literature.

The sensitivity to SEUs of SRAM-based FPGA systems can be analysed exploiting four main approaches: accelerated radiation ground testing, fault injection boards, analytical computation, and fault simulation.

4.1.1 Radiation Testing

Accelerated radiation ground testing [28, 44, 46, 79, 80] aims at emulating the effects of SEUs by exposing a prototype of the FPGA-based system to a flux of radiations, originated by either a radioactive source or a particle accelerator. While being exposed to the radiations flux, the prototype is fed with a set of input patterns, and its behaviour is monitored.

Advantages of radiation testing experiments to assess the sensitivity to SEUs of a digital system are:

- The experiments are carried out using a prototype of the system.
- The prototype of the system can be exposed to a radiation environment that can accurately emulate the environment in which the system will work.

Because of these two points radiation testing gives accurate results. Drawbacks of these techniques are:

- The impossibility of injecting SEUs only in the configuration memory of the FPGA, since the whole chip area will be irradiated (including user resources).
- A possibility that the device be permanently damaged after the experiment.
- High cost.

4.1.2 Fault Injection

Several injection boards have been developed in order to evaluate the impact of SEUs in the configuration memory of circuits mapped on SRAM-based FPGAs [5, 8, 178, 191]. These boards emulate the occurrence of SEUs by modifying the bitstream of the target system whose dynamic behavior is then evaluated. Fault injection can be performed either before downloading the bitstream on the device under test, or at run time exploiting partial dynamic reconfiguration. Unlike radiation testing experiments, fault injection makes it possible to focus the analysis on SEUs in the configuration memory of the FPGA, leaving out any other resources. Moreover fault injection avoids the risk of damaging the device under analysis. The major drawbacks of SEU injection boards are high costs, complex usability, and chip and vendor dependence. Moreover both fault injection and radiation testing have an additional drawback: they are applied late in the design process, only when a physical prototype of the system is available, thus modifications of the systems may be expensive and may take a long time.

4.1.3 Analytical Methods

Analytical approaches, such as reported in [24, 100, 176, 177] have been developed to avoid the high cost of radiation testing and the long experimental time of fault injection. In [176] and [177], a model based on the structure of the design implemented on the FPGA is built, and the topological modifications induced by SEUs in each configuration bit are deduced, thus discovering which SEUs affect the design. In [24], sensitive paths to SEUs are identified by combining the error probability of all nodes of the circuit with the error propagation probability of each path of the circuit. Finally in [100] an accurate probabilistic model to estimate the reliability of SRAM-based FPGA system is presented. Given the probability of occurrence of a SEU, the model is able to estimate the probability of having a system failure after a given amount of time. The drawback of these approaches is that, since the analysis is carried out without respect to the input patterns fed into the system, they are able to provide a worst case analysis while they cannot give information about the behaviour of the system in its normal operating conditions.

4.1.4 Fault Simulation

Although a large number of fault simulators for digital circuits can be found in the literature, very few simulation approaches targeting the analysis of the effects of SEUs have been proposed. Moreover, if we look for simulators that specifically address the FPGA technology we find an even smaller number of works. In [172, 43] two simulators of SEUs affecting digital circuits have been proposed. Both simulators work at the gate-level representation of the circuit, thus ensuring accurate results, but both do not take into account any details specific of the FPGA technology. To the best of our knowledge, the only simulator targeting SEUs in FPGAs is *SST* [94]. SST is a

set of TCL scripts able to modify the HDL description of the circuit in order to emulate the effects of SEUs, and then to interact with standard RTL-simulators, such as ModelSim [130]. Since SST works on the RTL representation of the system, it is just able to emulate the effects of SEUs in user resources, e.g., flip-flops and memories, but it is not able to reproduce the effects of SEUs in the configuration memory. To the best of our knowledge no simulators able to reproduce the effects of SEUs in the configuration memory of SRAM-based FPGA systems has been proposed, apart from ASSESS, that was preliminary presented in [31, 32, 34, 35]. .

4.2 The SAN Formalism and the Möbius tool

Stochastic Activity Networks [170] are an extension of the Petri Nets (PN) formalism [149]. SANs are directed graphs with four disjoint sets of nodes: *places*, *input gates*, *output gates*, and *activities*. The latter replace and extend the *transitions* of the PN formalism.

The topology of a SAN is defined by its input and output gates and by two functions that map input gates to activities and pairs (*activity*, *case*) (see below) to output gates, respectively. Each input (output) gate has a set of *input* (*output*) places. Each SAN activity may be either *instantaneous* or *timed*. Timed activities represent actions with a duration affecting the performance of the modelled system, e.g., message transmission time, recovery time, time to fail. The duration of each timed activity is expressed via a *time distribution* function. Any instantaneous or timed activity may have mutually exclusive outcomes, called *cases*, chosen probabilistically according to the *case distribution* of the activity. Cases can be used to model probabilistic behaviours, e.g., the failure probability of a component. An activity *completes* when its (possibly instantaneous) execution terminates.

As in PNs, the state of a SAN is defined by its *marking*, i.e., a function that, at each step of the net's evolution, maps the places to non-negative integers (called the *number of tokens* of the place). Whereas the PN formalism defines a fixed *enabling condition* to determine which transitions are enabled, and a fixed *firing rule* to determine the next marking after a transition has taken place, SANs enable the user to specify any desired enabling condition and firing rule for each activity. This is accomplished by associating an *enabling predicate* and an *input function* to each input gate, and an *output function* to each output gate. The enabling predicate is a Boolean function of the marking of the gate's input places. The input and output functions compute the next marking of the input and output places, respectively, given their current marking. If these predicates and functions are not specified for some activity, the standard PN rules are assumed.

The evolution of a SAN, starting from a given marking μ, may be described as follows: (i) The instantaneous activities enabled in μ complete in some unspecified order; (ii) if no instantaneous activities are enabled in μ, the enabled (timed) activities become *active*; (iii) the completion times of each active (timed) activity are computed

stochastically, according to the respective time distributions; the activity with the earliest completion time is selected for completion; (iv) when an activity (timed or not) completes, one of its cases is selected according to the case distribution, and the next marking μ' is computed by evaluating the input and output functions; (v) if an activity that was active in μ is no longer enabled in μ', it is removed from the set of active activities.

Graphically, places are drawn as circles, input (output) gates as left-pointing (right-pointing) triangles, instantaneous activities as narrow vertical bars, and timed activities as thick vertical bars. Cases are drawn as small circles on the right side of activities. Gates with default (standard PN) enabling predicates and firing rules are not shown.

4.2.1 The Möbius Tool

Möbius [63, 49] is a software tool that provides a comprehensive, easy-to-use and flexible graphical environment for model-based system analysis. The main features of the tool include: (i) multiple high-level modelling formalisms, including, among others, Stochastic Activity Networks (SANs) [170] and PEPA fault trees [93]; (ii) the possibility of extending the behaviour of a SAN model by attaching C++ functions to input and output gates. (iii) a hierarchical modelling paradigm, allowing one to build complex models by first specifying the behaviour of individual components and then by combining the components to create a model of the complete system; (iv) customised measures of system properties; (v) distributed discrete-event simulation, to evaluate measures using efficient simulation algorithms.

The Möbius tool introduces two extensions to the SAN formalism: *extended places* and *shared variables*. Extended places are places whose marking is a complex data structure instead of a non-negative integer. Shared variables are (possibly complex) data structures that are shared between different SANs, thus enabling them to communicate. Enabling predicates and input and output functions of the gates are specified as C++ code.

SAN models can be composed by means of *Join* and *Rep* operators. Join is used to compose two or more SANs. Rep is a special case of Join, and is used to construct a model consisting of a number of replicas of a SAN. Models composed with Join and Rep interact via *place sharing*.

Properties of interest are specified with *reward functions* [171]. A reward function specifies how to measure a property on the basis of the SAN marking. Measurements can be conducted at specific time instants, over periods of time, or when the system reaches a steady state.

4.3 The ASSESS Tool

ASSESS is simulator of SEUs affecting the configuration memory of SRAM-based FPGA systems. The simulator can be used for the analysis of the SEU sensitivity,

i.e., the probability of a system failure given that a configuration memory bit has been corrupted by a SEU, and for the assessment of the failure probability due to SEUs, i.e., the probability of a system failure given a probability of the occurrence of an SEU in the configuration memory of the system.

The ASSESS tool is a C++ program generated by the Möbius environment. The high level structure of ASSESS is shown in Figure 4.1. The simulator is composed of a *SEU Injector (SI)* module, an *Input Pattern Generator (IPG)* module, and a *Netlist Simulator (NS)* module.

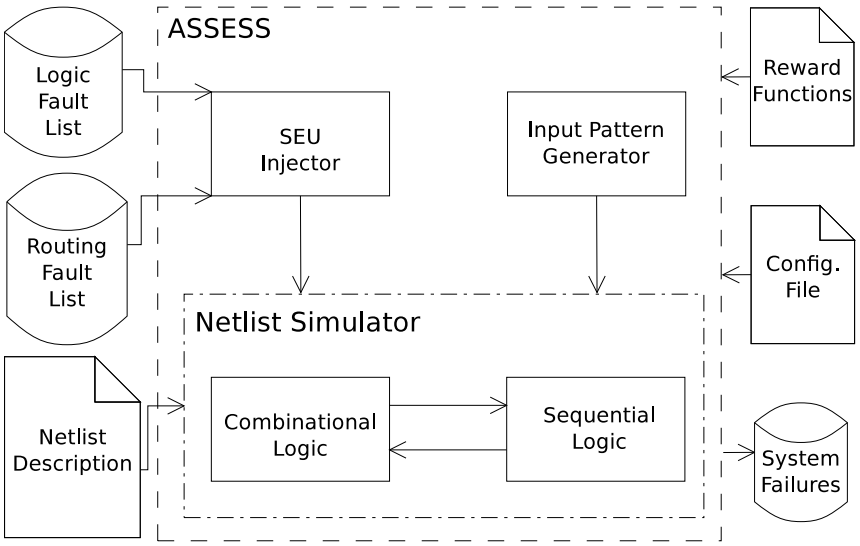


Figure 4.1. Flow Diagram of the Simulation Environment.

The SI module is in charge of injecting SEUs in the netlist during the simulation. The SI module is fed with two lists of SEUs: the list of SEUs in the memory bits controlling logic components and the list of SEUs in the memory bits controlling the routing structure. SEUs in the memory bits controlling logic components are simulated by modifying the functionality performed by the faulty component. SEUs in the memory bits controlling the routing structure are simulated according to the fault model discussed in Section 3.3.1. The SEU injection module can work either in deterministic or in stochastic mode. The deterministic SEU injection is used to assess the SEU sensitivity of the system. In the deterministic SEU injection SEUs are injected one at a time, at the beginning of the simulation, and the simulation end when every SEU in the fault lists has been injected. The stochastic SEU injection is used to assess the failure probability given to SEUs. The stochastic SEU injection allows to specify the maximum number of SEUs that may occur and the probability of SEU occurrence. At each clock cycle, if the maximum number of occurred SEUs has not yet been reached,

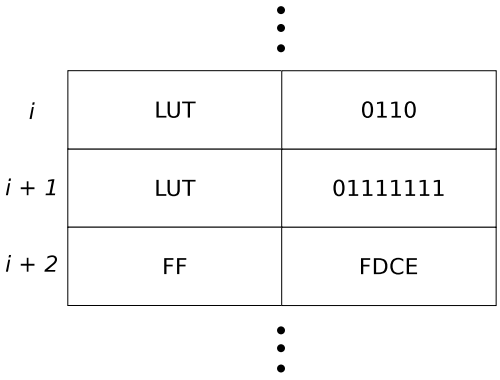
a SEU will be injected with the specified probability. The SI module is implemented by the `Fault Injection` SAN model (see Section 4.3.2).

The IPG module is in charge of generating the input patterns with which the NS module be fed. The IPG module can work either in deterministic or in stochastic mode. In the deterministic input pattern generation, the IPG module is provided with a list of input patterns. Moreover the IPG module working in deterministic mode can interact with external tools for the generation of test patterns, e.g., in [30] a genetic algorithm generated the test patterns that the IPG module fed to the simulation kernel. In the stochastic input pattern generation, the IPG module is provided with the signal probability of each input signal, i.e., the probability of a given input signal of assuming value 1 at a given clock cycle, and according with these probabilities it generates the input values for the input signals of the system. The IPG module is obtained by replicating I times the `Input Vector` SAN model (see Section 4.3.3), being I the number of input pins of the system.

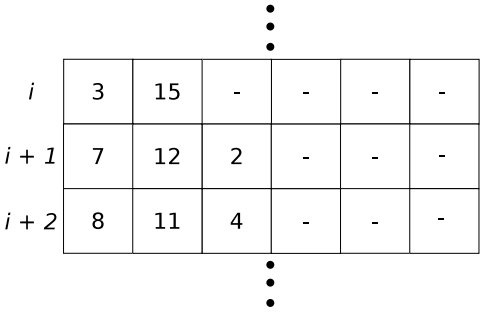
The NS module is composed of the Combinational Logic module and a Sequential Logic module. The Combinational Logic module simulates the behaviour of the combinational components of the FPGA system, i.e., LUTs, I/O buffers and multiplexers. The Sequential Logic module simulates the behaviour of the sequential components of the system, i.e., various types of flip-flops.

The functionality performed by each component is stored in a matrix called `Functions_Table` (shown in Figure 4.2(a)). The i^{th} entry of the matrix stores the type of the i^{th} component (LUT, flip-flop, buffer and multiplexer) and the specific functionality performed by the i^{th} component (logic function for LUTs, type of flip-flops, input/output for buffers). The example shown in Figure 4.2(a) reports about two LUTs performing the 2-input XOR and the 3-input OR and about a flip-flop with clear and clock-enable signals. All the simulated netlist components interact through a `Connectivity Matrix` (shown in Figure 4.2(b)) that simulates the connections among the components of the simulated system. The i^{th} entry of the matrix stores the ids of the component whose output represents the input of the i^{th} component. More in detail the j^{th} element of the entry of the matrix associated with the i^{th} component represents the id of the component whose output is connected to the j^{th} input pin of the i^{th} component. The example shown in Figure 4.2(b) reports that the output of components 3 and 15 are connected to the 1^{st} and the 2^{nd} input pins of the i^{th} component respectively. The Combinational and Sequential Logic modules and the functions table and connectivity matrix are instantiated in terms of functionalities performed by components and connections among them through the netlist description file.

Both the Combinational Logic module and the Sequential Logic modules are obtained by replicating C and S times the `Generic Component` SAN model (see Section 4.3.4), being C and S the number of combinational and sequential components of the system respectively.



(a) The functions table.



(b) The connectivity matrix.

Figure 4.2. Data structure of ASSESS for three components.

The simulation process is orchestrated by the `System Execution` SAN model (see Section 4.3.1), that coordinates the interaction of the various modules of the simulator. The basic functioning of ASSESS in the deterministic SEU injection mode is described by Algorithm 1.

The functioning of ASSESS in the stochastic SEU injection mode is described by Algorithm 2.

As shown in Figure 4.1 additional inputs of the simulator are: the *Reward Functions* defined on the netlist module (see Section 4.3.6) and a configuration file (see Section 4.3.7).

At the end of the simulation ASSESS produces the number of SEUs that caused a failure of the system. Additionally, by configuring dedicated simulation parameters, ASSESS can also generate the list of the applied input patterns and the detailed list of the SEUs that caused a failure of the system.

4.3.1 The `System Execution` Model

The `System Execution` SAN model (shown in Figure 4.3) orchestrates the execution of the of the simulation process by coordinating the interactions between the various

```
load the description of the netlist
load the list of SEUs in logic components
load the list of SEUs in routing components
load the list of unexcitable SEUs
system_failures := 0
critical_faults[] := {FALSE, ..., FALSE}
generate a random test pattern T
for every SEU s in the netlist do
   inject the i-th SEU
   apply test pattern T
   simulate the circuit
   if the system output is incorrect then
      system_failures := system_failures + 1
      critical_faults[i] := TRUE
   end if
   correct the i-th fault
end for
```

Algorithm 1: The simulation algorithm with deterministic SEU injection.

```
load the description of the netlist
load the list of SEUs in logic components
load the list of SEUs in routing components
load the list of unexcitable SEUs
system_failure := false
injected_SEUs := 0
generate a random test pattern T
apply test pattern T
for every clock cycle do
   if injected_SEUs < N_MAX_SEUs then
      randomly decide whether as SEU has to be injected
      if an SEU has to be injected then
         randomly select a SEU
         inject the selected SEU
         injected_SEUs = injected_SEUs + 1
      end if
   end if
   simulate the circuit
   if the system output is incorrect then
      system_failure := true
   end if
end for
```

Algorithm 2: The simulation algorithm with stochastic SEU injection.

modules of the simulator. Places `Input_Lines`, `Internal_Lines` and `Output_Lines` model the input, internal and output signals respectively. Place `Expected_Output` models the expected output for the unfaulty system. Place `n_clock_cycles` models the number of simulated clock cycles in a given time instant of the simulation. Finally `end_fault_injection` is a place with which the SEU Injector module informs the `System Execution` model that the last SEU was injected. Places `Internal_Lines` and `Output_Lines` and `Expected_Output` are shared between the `System Execution` model and the Netlist Simulator module. Place `Input_Lines` is shared among the `System Execution` model, the Input Pattern Generation module and the Netlist Simulator module. Place `n_clock_cycles` is shared between the `System Execution` model and the Input Pattern Generation module. Finally place `end_fault_injection` is shared between the `System Execution` model and the SEU Injector module. `System Execution` is structured as a loop consisting of the following five steps:

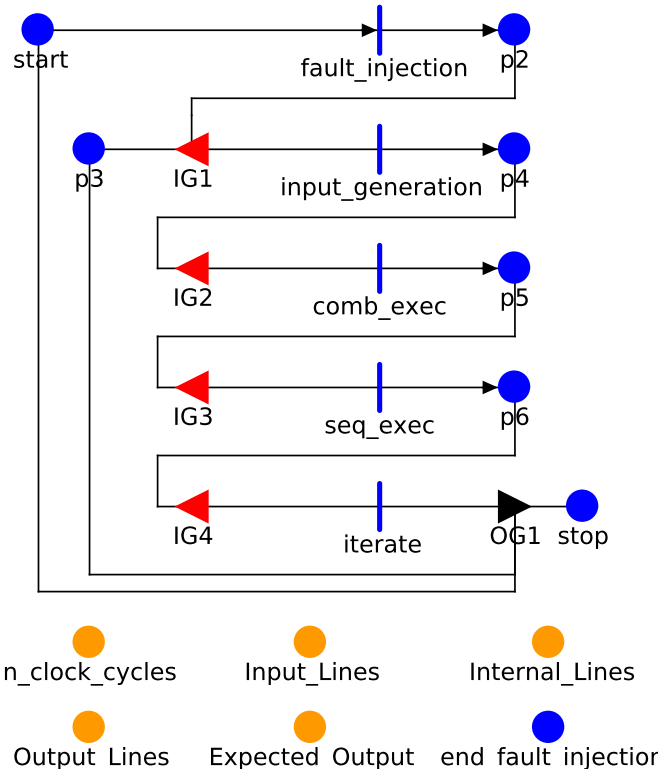


Figure 4.3. The System Execution SAN model.

1. When place `start` contains a token (at the beginning of the simulation), activity `fault_injection` becomes active. When the execution of this activity termi-

nates, a token is moved into place p2, thus activating the execution of the SEU Injector module.

2. At the end of the execution of the SEU Injector module, gate IG1 enables activity input_generation. When the execution of this activity ends, a token is moved into place p3, and the execution of the Input Pattern Generation module starts.
3. When the execution of the Input Pattern Generation module terminates, gate IG2 enables activity comb_exec. When the execution of this activity terminates, a token is moved into place p4, and the Combinational Logic module is activated.
4. When the Combinational Logic module terminates, gate IG3 enables activity seq_exec. When the execution of this activity terminates, a token is moved into place p4, and the Sequential Logic module becomes active.
5. When the execution of the Sequential Logic module ends, gate IG4 enables activity iterate. When the execution of this activity terminates, a token is moved into place p5, thus activating the execution of gate OG1. After increasing the contents of place n_clock_cycles, OG1 performs the following checks: (i) If deterministic SEU injection was selected:
   - if the value stored in n_clock_cycles does not equal the required number of simulated clock cycles, OG1 moves a token into place p3, thus reactivating Input Pattern Generation module.
   - if the value stored in n_clock_cycles equals the required number of simulated clock cycles and the content of place end_fault_injection is 0 (meaning that at least a SEU has still to be injected), OG1 resets the contents of n_clock_cycles and moves a token into place start, thus reactivating the SEU Injector module.
   - if the value stored in n_clock_cycles equals the required number of simulated clock cycles and the contents of place end_fault_injection is 1 (meaning that all SEUs have already been injected), OG1 moves a token into place stop, thus ending the simulation

   (ii) If stochastic SEU injection was selected:
   - if the value stored in n_clock_cycles does not equal the required number of simulated clock cycles, OG1 moves a token into place start, thus reactivating the SEU injection module.
   - if the value stored in n_clock_cycles equals the required number of simulated clock cycles OG1 moves a token into place stop, thus ending the simulation

4.3.2 The Fault Injection Model

The Fault Injection SAN model (shown in Figure 4.4) implements the SEU Injector component presented in the high level description of ASSESS. Place injected_fault is shared between the SEU Injector module and the Netlist Simulator module and represents the ID of the injected fault. Place n_injected_faults represents the number

of SEUs currently injected in the circuit, while place `stochastic_deterministic` specifies whether stochastic or determinist SEU injection is required. The detailed behaviour of the model is described by the following steps:

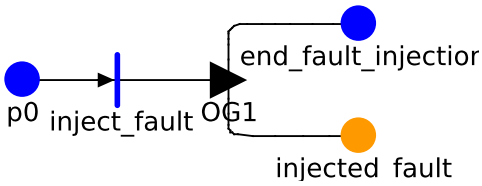


Figure 4.4. The Fault Injection SAN model.

1. When place `start` contains a token (when the `System Execution` model activates the injection of a fault), activity `inject_fault` becomes active.
2. When activity `inject_fault` terminates, gate `OG1` is executed. This gate models the actual injection of a SEU by storing the two lists of faults (SEUs in logic and routing components) and by specifying the ID of the injected fault in place `injected_fault`. If deterministic SEU injection is required `OG1` performs the following steps:
   a) Inject a SEU in the circuit.
   b) If the injected SEU is the last fault of the two lists of SEUs, move a token in place `end_fault_injection` in order to inform the `System Execution` module that the fault injection process is terminated.

   Conversely, if stochastic SEU injection is required `OG1` performs the following steps:
   a) If the required maximum number of injected SEUs has been reached, do nothing.
   b) Otherwise randomly chose an SEU from the two fault lists, inject it in the circuit and increment the content of place `injected_faults`.

4.3.3 The `Input Vector` Model

The `Input Vector` SAN model (shown in Figure 4.5) is the basic building block of the Input Pattern Generator component of ASSESS. In particular for each input pin of the netlist a replica of the `Input Vector` module is instantiated. Each replica is uniquely identified by the ID of the associated input pin stored in place `InputPinID`. The purpose of each replica of the model is to generate (either deterministically or stochastically) the input signal applied to the associated input pin p at the clock cycle c specified by the content of place `n_clock_cycles`. The detailed behaviour of the model is described by the following steps:

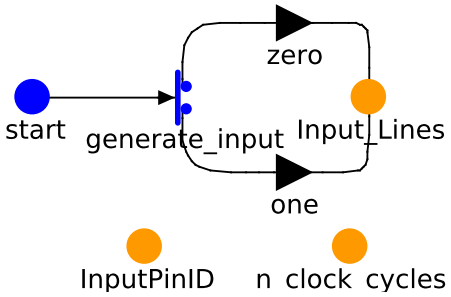


Figure 4.5. The Input Vector SAN model.

1. When place `start` contains a token (when the `System Execution` module activates the generation of an input pattern), activity `generate_input` becomes active. If the user required the stochastic input pattern generation, activity `generate_input` stochastically activates either gate `zero` or gate `one` according to the input signal probability for input pin p at clock cycle c specified in the simulation configuration file (see Section 4.3.7). Similarly, if the user required the deterministic input pattern generation, activity `generate_input` activates either gate `zero` or gate `one` according to the logical value for input pin p at clock cycle c specified in the simulation configuration file (see Section 4.3.7).
2. When activity `generate_input` terminates, either gate `zero` or gate `one` is executed. If gate `zero` is executed, the i^{th} location of place `Internal_Lines` (which is an array of Boolean) is set to a logical 0, to a logical 1 otherwise.

4.3.4 The `Generic Component` Model

The `Generic Component` SAN model (shown in Figure 4.6) is the basic building block of the Netlist Simulator component of ASSESS. `Generic Component` is a customizable SAN model able to model all the combinational and sequential components of the netlist. In particular, for a given system, a replica of `Generic Component` is instantiated for each component of the netlist. Each replica is uniquely identified by an ID stored in place `ComponentID`. The specific behaviour of the netlist component is modeled by a C++ function, attached to gate `OG1`, that simulates the logic behaviour of the component itself. The components that can be simulated are: (i) Input/Output buffers; (ii) lookup tables; (iii) multiplexers; and (iv) various types of flip-flops (normal D flip-flops, with clock-enable, with asynchronous reset and set). The detailed behaviour of the model is described by the following steps:

1. When place `start` contains a token (when the `System Execution` module activates the execution of the combinational or sequential components), activity `exec_component` becomes active.
2. When activity `exec_component` terminates, gate `OG1` is executed. This gate models the actual logic behaviour of the component by executing the attached C++

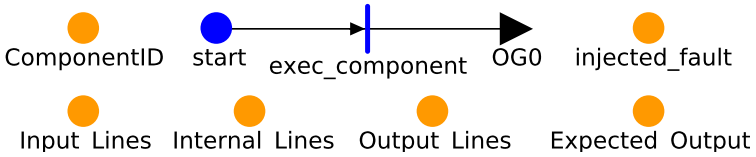


Figure 4.6. The Generic Component SAN model.

function. More precisely, for a given component i, the function F specified in gate `OG1` retrieves the logic function f associated with component i. Then F retrieves the inputs of component i from places `Input_lines` and `Internal_Lines`. F calculates the expected output of component i by applying f to the input signals of i and then stores the calculated value in the i^{th} location of place `Expected_Output` (which is an array of Boolean). Then, if component i is affected either directly (SEUs in configuration bits controlling logic resources) or indirectly (SEUs in configuration bits controlling routing resources) by the fault identified by the ID specified in place `injected_fault` (which is shared with the `Fault Injector` module), F calculates the actual output of the component and stores it the i^{th} location of place `Output_Lines`.

4.3.5 The Composition of the SAN Models

The structure of the composition of the SAN models of which ASSESS is composed is shown in Figure 4.7. The upper hierarchical level is constituted by the join between the `System_Execution` model, the `Fault_Injection` model and `Combinational_Logic`, `Sequential_Logic` and `Input_Pattern_Generator`.

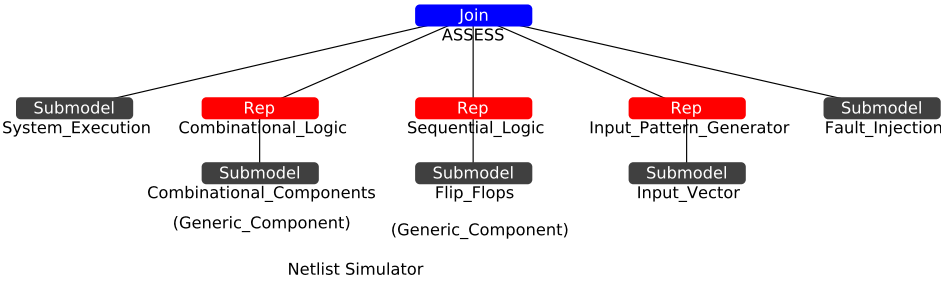


Figure 4.7. The composition of the SAN models of the ASSESS tool.

`Combinational_Logic` and `Sequential_Logic` constitute the netlist simulator of ASSESS. Both `Combinational_Logic`, `Sequential_Logic` are the replication of the basic SAN model `Generic_Component`. In particular `Combinational_Logic` models the combinational components of the netlist and is composed of a number of replicas of `Generic_Component` equal to the number of LUTs, multiplexer and I/O buffers. Similarly `Sequential_Logic` models the sequential components of the netlist and

is composed of a number of replicas of `Generic_Component` equal to the number flip-flops. The `Input_Pattern_Generator` is composed of the replicas of the basic `Input_Vector` equal to the number of input pins of the netlist.

4.3.6 The Reward Function

A reward model was added to the model of the netlist in order to make possible the assessment of the sensitivity to SEUs of the system under analysis. In particular we were interested in discovering if the output of the system was different from the expected output for at least one clock cycle during the simulation. To this purpose the designed reward function was the one shown by Algorithm 3:

```
if (System_Execution→p5 == 1) then
  for (int i = 0; i < N_OUTPUT_PINS; ++i) do
    if (System_Execution→Output_Lines [i] !=
        System_Execution→Expected_Output[i]) then
      return 1
    end if
  end for
end if
```

Algorithm 3: The reward function.

This reward function returns 1 whenever a clock tick arrives and the actual output of the system differs from the expected output for at least one output signal.

Thus at the end of an exhaustive fault simulation this reward function returns the number of SEUs that caused a failure of the system, thus allowing the assessment of the criticality of SEUs and of the sensitivity to SEUs of the system.

When the stochastic fault injection is selected by the user, a number of simulation runs will be performed. At the end of these simulations the reward function returns the number of runs in which the actual output and the expected output where different in at least one clock cycle, thus allowing to assess the failure probability of the system.

4.3.7 Building and Configuring ASSESS

The Möbius tool generates a C++ implementation of the composed SAN models, together with its reward function. More in detail, the simulator is obtained through the following steps:

1. Describing the SAN models of the various basic components composing the simulator.
2. Describing the SAN model that specifies the interaction among the basic components.

3. Specifying which basic components have to be replicated and how many replicas have to be generated.
4. Specifying the reward function that has to be executed on the simulated model.
5. Having the Möbius tool generate the executable.

Before running ASSESS, the following simulation parameters have to be initialized through a configuration file:

- `N_CLOCKS`: the number of clock cycles the user wants to simulate.
- `RANDOM_INPUTS`: `TRUE` if the user requires random input pattern generation; `FALSE` otherwise.
- The input signal probabilities if random input pattern generation is required by the user; the required deterministic input patterns otherwise.
- `RANDOM_SEU_INJECTION`: `TRUE` if the user requires random SEU injection; `FALSE` otherwise.
- `N_MAX_SEUs`: the maximum number of injected SEUs (if stochastic SEU injection is required by the user).
- The SEU occurrence probability if random SEU injection is required by the user.
- `SAVE_INPUTS`: `TRUE` if the user requires that ASSESS saves the applied input patterns in an additional output file (this option is useful when random input pattern generation is required by the user); `FALSE` otherwise.
- `SAVE_FAULTS`: `TRUE` if the user requires that ASSESS produces a detailed report containing the list of SEUs that caused a failure of the system; `FALSE` otherwise. Note that if `SAVE_FAULTS` is set to `FALSE` ASSESS will only produce as output the number of SEUs that caused a failure of the system.

5

The UA^{2}TPG Tool

5.1 Related Work: Techniques for Fault Untestability Analysis

A number of works addressing various aspects of the analysis of untestability of faults in digital systems can be found in the literature. In [151] and [153] a new subclass of untestable faults, called *register enable stuck-on* is defined and a method for generating property specification language (PSL) assertions for proving the untestability of this class of faults is presented. In these papers stuck-at faults on the clock-enable signals of registers at the register transfer level (RTL) are addressed. The same authors propose in [152] a hierarchical untestability identification method. The method addresses untestable faults in functional units, such as adders and multiplexers, at the RTL level.

In [182] a preprocessing method for accelerating SAT-based ATPGs by eliminating untestable faults is presented. The method takes into account the stuck-at fault model and it addresses only *easy-to-classify* untestable faults.

In [125] two algorithms (FILL and FUNI) for untestability demonstration of stuck-at faults are presented. FILL identifies large subsets of illegal states in synchronous sequential circuits, and FUNI finds untestable faults that require illegal states previously found by FILL to be detected.

Apart from UA^{2}TPG, the only works in the literature addressing the analysis of the untestability of SEUs in the configuration memory of SRAM-based FPGAs were presented in [33] and in [36]. In these works SEUs in the configuration bits controlling logic and routing resources respectively were considered. Both techniques performed only a partial untestability analysis, since they allowed to discover those SEUs that could not be excited by any configuration of the inputs of the system, but both were not able to analyse the problem of the unpropagability of faults. Given this, to the best of our knowledge, UA^{2}TPG represents the first tool able to determine which SEUs in the configuration bits actually used by a given FPGA-based system are not testable.

5.2 The SAL Environment

The Symbolic Analysis Laboratory (SAL) is a framework for abstraction, program analysis, theorem proving and model checking of concurrent systems expressed as transition systems through a specification language [29].

The main tools available in the environment are a SAL to Java compiler for the simulation of specifications, the SAL-SMC model checker for property verification, an automatic translator from SAL to PVS specifications for theorem proving, an invariant generator and a counterexample generator.

In the following we introduce the basics of the SAL language, of the SAL model checker and of the LTL logic that have been used to model netlists and to specify and to to prove SEU untestability theorems.

5.2.1 The SAL Language

The core of SAL is the specification language used to describe concurrent systems. The language is also used as an common description language used by the analysis tools of the SAL environment.

The SAL language is a strongly-typed description language. Supported types are: booleans, scalars, integers and integer subranges, records, arrays and abstract datatypes. Expressions consist of constants, variables, applications of Boolean, arithmetic, bit-vector operations and array and record selection and update. Declarations of new types is also allowed by the language. Conditional expressions and user-defined functions are also supported.

The basic concept in the SAL specification language is the `Module`. A SAL module is a self-contained specification of a transition system. A module consists of a `State` and an `Initialization` on the state and a list of `Transitions` on the state.

The state is defined by four disjoint sets of `Input`, `Output`, `Global` and `Local` variables. The input and global variables are *observed*, in the sense that their value can be just read. The output and local variables are *controlled*, in the sense that their value can be both read and written. Each SAL variable has two values, the *current* value (denoted, e.g., by `x`) and the *next* value (denoted, e.g., by `x'`) valid in the current and the next state (respectively) of the module.

The initialization is used to specify an initial value for all or some of the controlled state variables of the module. An initialization is simply specified as an assignment between the variable and the result of an expression, as follows:

variable = expression

The transitions of a module can be specified *variable-wise*, by means of `Definitions`, or *transition-wise* by means of `Transitions`. A definition is a simple assignment between a controlled variable and the result of an expression.

Transitions are assignments between next-state variables and the result of expressions. A transition is of the form:

variable' = expression

A list of transitions can be specified as a `Guarded Command`. A guarded command is composed of a guard, i.e., a boolean condition defined on state variables, and one or more transitions. The guard has to be satisfied in order to perform the transitions. A guarded command is of the form:

$$\begin{aligned} &\textit{condition} \rightarrow \textit{variable}_1\text{'} = \textit{expression}_1 \\ &\qquad\vdots \\ &\qquad\textit{variable}_n\text{'} = \textit{expression}_n \end{aligned}$$

where *condition* is the guard and $\textit{variable}_1\text{'} = \textit{expression}_1 \cdots \textit{variable}_n\text{'} = \textit{expression}_n$ is the list of transitions.

The SAL language provides the `IN` construct that denotes nondeterministic choice among a set of values. However in some contexts the language uses `IN` also for deterministic assignment. The SAL language allows the composition of different modules. Modules can be combined by either synchronous (`||`) or asynchronous (`[]`) composition. Several modules can be collected in a SAL `Context`. Contexts may also include constants, types declarations and theorems.

5.2.2 The SAL Model Checker

The SAL-SMC (Symbolic Model Checker) uses LTL (Linear Temporal Logic) as assertion language [162]. LTL formulas state properties about each linear path induced by a transition system. Typical LTL operators are:

- `G(p)` states that p is always true.
- `F(p)` states that p will be eventually true.
- `U(p,q)` states that p is true until a state is reached where q is true.
- `X(p)` states that p is true in the next state.

For a formal definition of LTL see [162]. Typical properties expressed with LTL formulas are *safety*, in the form $\texttt{G}(\neg\chi)$, stating that the undesired condition χ is never satisfied, and *liveness*, in the form $\texttt{G}(\texttt{F}(\psi))$ or $\texttt{G}(\gamma \rightarrow \texttt{F}(\psi))$, stating that the desired condition ψ will be eventually satisfied or that the desired condition ψ will be eventually satisfied if condition γ is satisfied.

SAL-SMC follows the automata-theoretic approach [185], in which the complemented LTL formula and the transition system are translated into Büchi automata and analysed as Binary Decision Diagrams (BDDs). Given an LTL formula ϕ, one can build a Büchi automaton A_ϕ associated with ϕ such that the set of words accepted as input by A_ϕ is identical to the set of computations that satisfy ϕ.

In particular given a transition system S and an LTL formula ϕ, to check whether ϕ is satisfied on S the following steps are performed: (i) The Büchi automaton A_S associated with system S is built; (ii) the LTL formula ϕ is negated and then the Büchi

automaton $A_{\overline{\phi}}$ associated with the negated formula is built; (iii) A_S and $A_{\overline{\phi}}$ are represented as BDDs and (iv) the intersection between the BDD associated with A_S and the one associated with $A_{\overline{\phi}}$ is calculated. If the intersection is empty, then ϕ is satisfied on S, otherwise the LTL formula is not satisfied on the transition system, and a counterexample is provided by the model checker.

5.3 The Untestability Analysis and ATPG Tool

The proposed tool relies on the SAL description language to describe the structure of the netlist under analysis, on the LTL to specify the untestability theorems and on SAL-SMC to prove the untestability theorems and to generate counterexamples that will be used to extract test patterns to detect the testable SEUs..

In the following we first show how to use the SAL language to model netlists of FPGA-based systems and SEUs occurring in the configuration memory of the device, we then show how to write untestability theorems using LTL and finally we illustrate the execution flow of the tool.

5.3.1 Modeling SRAM-based FPGA Netlists

We used the language provided by SAL to model FPGA-based systems starting from a description of the circuit at the netlist level before the place&route phase. Each netlist is described by a SAL `MODULE`. Each component in the netlist is modeled as a SAL `LOCAL` Boolean variable that represents the output of the component itself. Input and output pins of the system are modeled by SAL `INPUT` and `OUTPUT` Boolean variables. In particular the behaviour of output pins is modeled as an assignment between an output variable and the local variable modeling the associated output buffer (see below).

The behaviour of asynchronous components is described by `Definitions`. The behaviour of synchronous components is described by `Transitions`. To show how we modeled the behaviour of components we refer to the simple example of netlist shown in Figure 5.1.

The behaviour of an input buffer can simply be described as an assignment between a local variable, modeling the buffer, and an input variable, modeling the associated input pin. Similarly the behaviour of an output buffer can be described as an assignment between two local variables, one modeling the buffer and the other modeling the connected component. Examples of input and output buffers are shown below:

```
i_buff_0=i_pin_0;
i_buff_1=i_pin_1;
o_buff_0=d_ff_0;
```

The behaviour of LUTs is described by the corresponding logic functions. The lookup tables of the circuit in Figure 5.1 can be modeled as follows:

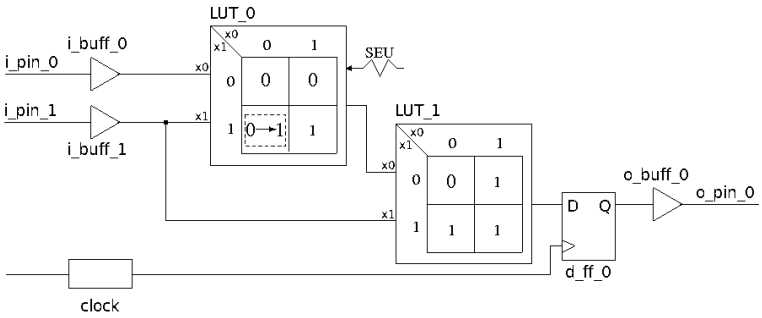


Figure 5.1. An example netlist.

```
LUT_0=i_buff_0 AND i_buff_1;
LUT_1=LUT_0 OR i_buff_1;
```

Note that `LUT_0` and `LUT_1` perform the `AND` and the `OR` function respectively. Multiplexers can be described by an `IF THEN ELSE` clause, as follows (s represents the select signal for the multiplexer):

```
y IN IF(s=FALSE)
       THEN {x1}
     ELSE {x2}
     ENDIF;
```

Flip-flops are described by `Transitions`. D-flip-flops can be described as a simple assignment between the next value of the flip-flop and the current value of its input:

```
d_ff_0'=LUT_1;
```

Other types of flip-flops (FDC, FDP, FDCE and FDPE) can be described by an `IF THEN ELSE` clause, as follows (e represents the clock enable signal, c the clear signal and p the pre-set signal):

```
q' IN IF(c=TRUE)
        THEN {FALSE}
      ELSE {d}
      ENDIF;

q' IN IF(p=TRUE)
        THEN {TRUE}
      ELSE {d}
      ENDIF;

q' IN IF(c=TRUE)
        THEN {FALSE}
      ELSIF (e=FALSE)
        THEN {q}
      ELSE {d}
```

```
        ENDIF;

q' IN IF(p=TRUE)
        THEN {TRUE}
      ELSIF (e=FALSE)
        THEN {q}
      ELSE {d}
      ENDIF;
```

5.3.2 Modeling SEUs Affecting the Configuration Memory

We define an untestable SEU as an SEU that is not able to corrupt the output of the systems even if it is able to modify the structure of the system (in terms of functionalities performed by one of its LUTs or in terms of connections among components of the device). In other words given a system T performing function F and an SEU s occurring in the configuration memory of T, and modifying the function performed by T from F into F^*, we say that s is untestable if the output of F always equals the output F^* being F and F^* fed with the same inputs.

We model the effects of SEUs affecting the configuration bits associated with logic components by modifying the functionality performed by the component according with the corrupted configuration bit. As an example, with reference to Figure 5.1, we model the effect of the SEU shown in the figure by modifying the logic functionality of `LUT_0` from $f =$ `i_buff_0 AND i_buff_1` to $f^* =$ `i_buff_1`.

Similarly we model SEUs affecting configuration bits controlling routing resources: with reference to Figure 3.9, let P_i and P_j be two PIPs in a switchbox, H and C two components connected through P_i, with signals going from H (*source*) to C (*destination*), and let K and D be two components similarly connected through P_j. Further, let $\hat{H}$ and $\hat{K}$ be the SAL variables modeling H and K respectively, $f_{\hat{H}}$ and $f_{\hat{K}}$ the logic functions implemented by H and K respectively and x_H and x_K the configuration of the input signals of H and K respectively. According to the caused logical effect, we model a SEU s in the configuration bits controlling P_i and P_j as follows (let us call H and K the *affected components*):

- An SEU causing a stuck-at 0 (1) on P_i is modeled by changing the function performed by $\hat{H}$ to
  $f^*_{\hat{H}} = FALSE\ (TRUE)$.
- An SEU causing a bridge between P_i and P_j is modeled by changing the function performed by $\hat{H}$ to
  $f^*_{\hat{H}} = f_{\hat{K}}(x_{\hat{K}})$
  and the function performed by $\hat{K}$ to
  $f^*_{\hat{K}} = f_{\hat{H}}(x_{\hat{H}})$.
- An SEU causing a Wired-AND (Wired-OR) between P_i and P_j is modeled by changing the function performed by $\hat{H}$ and by $\hat{K}$ to:
  $f^*_{\hat{H}} = f^*_{\hat{K}} = f_{\hat{H}}(x_{\hat{H}}) \wedge (\vee) f_{\hat{K}}(x_{\hat{K}})$

- An SEU causing a Wired-MIX between P_i and P_j is modeled by changing the function performed by $\hat{H}$ to:
  $f^*_{\hat{H}} = if(f_{\hat{H}}(x_{\hat{H}}) = f_{\hat{K}}(x_{\hat{K}}))$
  $then\ f_{\hat{H}}(x_{\hat{H}})$
  $else\ TRUE$
  and the function implemented by $\hat{K}$ to
  $f^*_{\hat{K}} = if(f_{\hat{H}}(x_{\hat{H}}) = f_{\hat{K}}(x_{\hat{K}}))$
  $then\ f_{\hat{K}}(x_{\hat{K}})$
  $else\ FALSE$

As we said in Chapter 3, a given SEU in a configuration bit controlling a PIP can propagate through a number of routing segments with different logical effects. Thus, a SEU having multiple propagation points is modeled by modifying the functions performed by all the affected components, according with the logical effects associated with the SEU.

5.3.3 Identifying Untestable SEUs

In order to analyse the untestability of a given SEU, according to what has been introduced in the previous sections, we build the SAL model of the unfaulty circuit and of the faulty one, we connect them to the same inputs and we check whether the outputs of the two systems are always the same or not.

We define untestability theorems as LTL safety formulas, in the form $G(\neg(\forall_i\ (O_i \neq {O_i}^*)))$ where O_i are the outputs of the unfaulty circuit and ${O_i}^*$ are the outputs of the faulty circuit. Such formulas simply state that it is always false that the output of the correct system is different from the output of the faulty circuit. If for a given SEU the theorem is proved, the SEU is demonstrated to be untestable.

In order to show a complete example we list the SAL code for the analysis of the untestability of the SEU shown in Figure 5.1 (we show the code for the unfaulty circuit, the code for the faulty circuit associated with the fault shown in the figure and the code for the untestability theorem).

```
untest : CONTEXT =
  BEGIN
    untest_circuit : MODULE =
    BEGIN
      % Input Pins (for both the unfaulty
      % and the faulty circuit)
      INPUT i_pin_0 : BOOLEAN
      INPUT i_pin_1 : BOOLEAN

      % Specification of the unfaulty circuit
      LOCAL i_buff_0_C : BOOLEAN
      LOCAL i_buff_1_C : BOOLEAN
      LOCAL LUT_0_C : BOOLEAN
```

```
        LOCAL LUT_1_C : BOOLEAN
        LOCAL d_ff_0_C : BOOLEAN
        LOCAL o_buff_0_C : BOOLEAN
        OUTPUT o_pin_0_C : BOOLEAN
        DEFINITION
          i_buff_0_C=i_pin_0_C
          i_buff_1_C=i_pin_1_C
          LUT_0_C=i_buff_0_C AND i_buff_1_C
          LUT_1_C=LUT_0_C OR i_buff_1_C
          o_buff_0_C=d_ff_0_C
          o_pin_0_C=o_buff_0_C
        INITIALIZATION
          d_ff_0_C=FALSE;
        TRANSITION
          d_ff_0_C'=LUT_1_C;

        % Specification of the faulty circuit
        LOCAL i_buff_0_F : BOOLEAN
        LOCAL i_buff_1_F : BOOLEAN
        LOCAL LUT_0_F : BOOLEAN
        LOCAL LUT_1_F : BOOLEAN
        LOCAL d_ff_0_F : BOOLEAN
        LOCAL o_buff_0_F : BOOLEAN
        OUTPUT o_pin_0_F : BOOLEAN
        DEFINITION
          i_buff_0_F=i_pin_0_F
          i_buff_1_F=i_pin_1_F
          LUT_0_F=i_buff_1_F
          LUT_1_F=LUT_0_F OR i_buff_1_F
          o_buff_0_F=d_ff_0_F
          o_pin_0_F=o_buff_0_F
        INITIALIZATION
          d_ff_0_F=FALSE;
        TRANSITION
          d_ff_0_F'=LUT_1_F;
    END;
    % Untestability Theorem
    untestability : THEOREM
      untest_circuit |-
        G(NOT(o_pin_0_C/=o_pin_0_F));
END
```

Note that, as expected, the correct and faulty circuits differ only for the function implemented by `LUT_0`, which is the component suffering from the SEU considered in the example.

5.3.4 Generating Test Patterns for Testable SEUs

When the SAL-SMC model checker is asked to demonstrated a theorem, if the theorem is not proved, a counter-example is automatically produced by the model-checker

itself. The counter-example provided by SAL-SMC after trying to prove an untestability theorem, defined according to what has been discussed in the previous section, is a sequence of input vectors applied to the inputs of the system that caused the theorem not to be proved. In particular the sequence of input vectors caused the output of the faulty system to be different from the output of the unfaulty one. Thus the produced sequence of input patterns represents a test pattern able to test the SEU under analysis.

5.3.5 The Execution Flow

The overall execution flow of UA^{2}TPG is shown in Figure 5.2. The Netlist Description File contains a simple description of the netlist in terms of functionalities performed by components and connections among components. The Logic Fault list contains the list of the faulty LUT functions associated with each SEU in each LUT actually used by the system. The Routing Fault list contains the list of the of the effects of each SEU in each PIP actually used by the system.

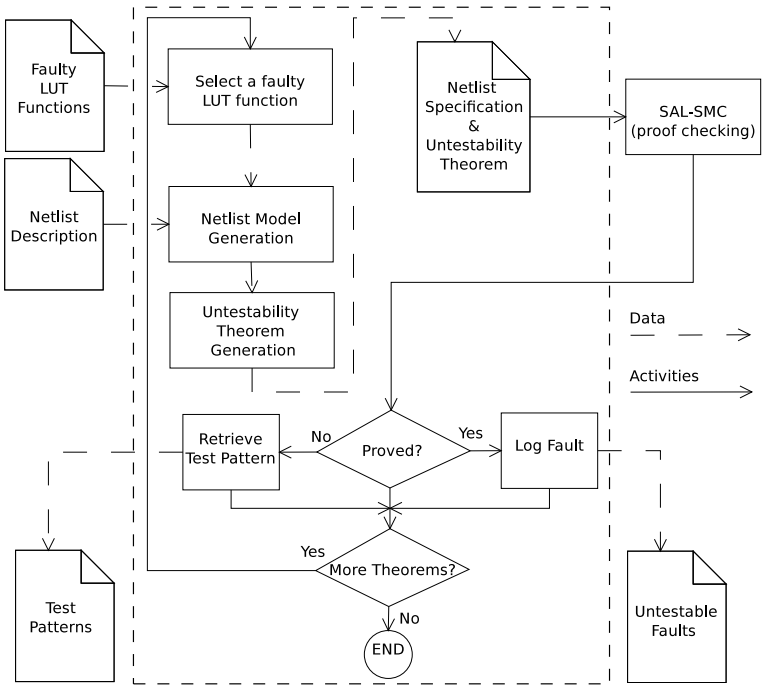


Figure 5.2. The data flow of the proposed tool.

For each SEU S_j that may occur in the configuration memory actually used by the SRAM-based FPGA system under analysis the tool performs the following steps:

1. If S_j affects the configuration memory controlling a LUT, load the faulty LUT function of associated with S_j from the Logic Fault list. If S_j affects the configuration memory controlling a PIP, load the list of effects associated with S_j from the Routing Fault list.
2. Build the model of the unfaulty system (as described in Section 5.3.1) starting from the Netlist Description file.
3. Build the model of the faulty system (as described in Section 5.3.2) starting from the Netlist Description file and from the logical effect induced by the SEU under analysis identified at step 1.
4. Build the untestability theorem (as described in Section 5.3.3).
5. Invoke SAL-SMC on the untestability theorem.
   - If the untestability theorem is proved (thus S_j is untestable) then log S_j in the list of the untestable SEUs.
   - If the untestability theorem is not proved (thus S_j is testable) then extract the test pattern able to detect S_j from the counter-example provided by SAL-SMC (as described in Section 5.3.4) and log it into the list of test patterns.

At the end of the untestability analysis the list of the untestable SEUs and the list of test patterns that test all the testable SEUs are generated. The list of test patterns contains a test for each testable SEU. This list is then compressed by eliminating all the duplicated test patterns and all those test patterns that are prefix of longer ones.

6

The GABES Tool

6.1 Related Work: Testing Techniques

Many approaches to automatic test generation for digital circuits [4, 142] are found in the literature. A broad classification can be made between *deterministic* and *random* test generation methods. Deterministic methods are based on algorithms, such as the D-algorithm [159, 160], PODEM [85], or FAN [78], that rely on knowledge of the circuit structure to compute sets of test vectors that can detect all possible stuck-at faults. These techniques are generally able to generate test patterns with a high fault coverage and an optimized length, but they suffer from long execution times. Random methods [187] produce test vectors as pseudorandomly generated n-tuples of input values, thus requiring no knowledge of circuit structure. Random methods generate test vectors more quickly than deterministic methods, but need a large number of vectors to ensure a high probability of detecting all faults. Newer pseudo-random techniques use re-seeding and bit changing to improve fault coverage [111, 6]. Some algorithms, such as RAPS [84] and SMART [3], combine random techniques with structural information in order to improve the efficiency of randomly generated test sets.

Another way to improve the quality of randomly generated test sets is using *coverage-directed* generation [108, 169]. This is an iterative and evolutionary approach, where at each step the fault coverage of a group of tests is evaluated by simulation, and at the next step the group is transformed in order to improve fault coverage and other desirable properties. Many techniques and criteria can be used to generate new tests at each step. In particular, *genetic algorithms* [131, 81, 74] have proved to be effective.

Early applications of genetic algorithms to test pattern generation were presented by Saab *et al.* [168], Rudnick *et al.* [164], and Corno *et al.* [54]. In the last twenty years genetic algorithms have been proposed for many tasks in validation and testing of digital circuits. Genetic algorithms have been used for test pattern generation addressing hardware defects in digital circuits [168, 164, 54, 165, 174], for test program

generation addressing microprocessor defects [51, 52, 53, 57] and microprocessor functional validation [55].

In the area of FPGA testing, two families of methods may be distinguished: *application-independent* and *application-dependent* methods. Application-independent methods, such as those reported by Huang *et al.* [102], Renovell *et al.* [157], and Stroud *et al.* [179], aim at detecting structural defects due to the manufacturing process of the chip. These techniques are mainly performed by the chip manufacturer, and thus they are also known as *manufacturer-oriented* techniques. These methods are called application-independent because they target every possible fault in the device without any consideration of which parts of the chip are actually used by the given design and which parts are not. These techniques use multiple test configurations of the FPGA chip and the associated ad-hoc generated test patterns. Each test configuration is intended to test a set of the possible faults of the chip.

Conversely, application-dependent methods [163, 180, 30] address only those resources of the FPGA chip actually used by the implemented system. Since these techniques are applied by the user after the system design has been defined, they are also known as *user-oriented*. The basic idea behind this family of techniques is that very often an FPGA-based system uses only a subset of the resources provided by the FPGA chip. Therefore, demonstrating that the resources used by the implemented system are fault-free is sufficient to guarantee the correct operation of the system itself. Application-dependent methods have been proposed for in-service testing of both structural defects [163, 180] and SEUs [30].

6.2 Evolutionary Approaches

Many complex problems may be solved by *search* methods, i.e., procedures that look for a solution by trying out many attempts until a satisfactory result is obtained. Such an attempt might be, e.g., a sequence of moves in a game, a set of variable assignments to solve an equation, or a set of parameter values to optimize a function. Often more than one solution exists, and some solution may be better than others according to given criteria [81].

A GA is a search method based on the analogy with the mechanisms of biological evolution. GAs require that any solution to a given problem be *encoded*, i.e., represented as a sequence of symbols, that stands for a chromosome (a sequence of genes) in the biological analogy. A GA starts from an initial set (a *population*) of tentative solutions (called *chromosomes*), *selects* the best ones according to a problem-specific *fitness* function, and the selected chromosomes are *combined* and *mutated* to produce a new population. These operations have a degree of randomness, depending on probability distributions whose parameters can be tuned. The process is repeated until a termination criterion is met.

Figure 6.1 shows a general structure of a GA, via an activity diagram, where T identifies each step of the GA.

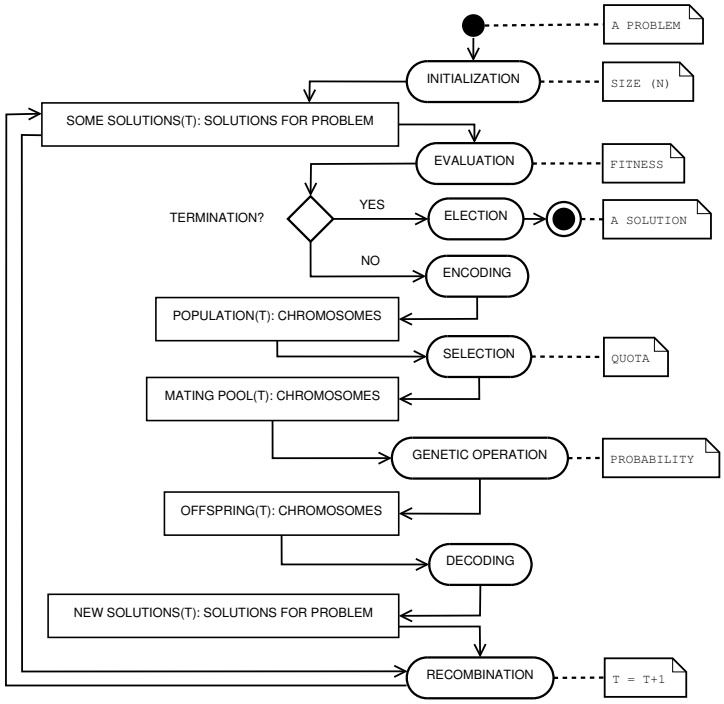


Figure 6.1. A scenario for the crossover operator.

More precisely, for a given optimization problem, an *initialization* process provides a set of randomly generated approximated solutions. Each solution is then *evaluated*, using an appropriate measure of fitness. If the *termination* criteria are satisfied, a solution is then *elected* as (sub)optimal for the problem. If not, each solution is encoded as a chromosome. The chromosomes evolve through successive generations, i.e., iterations of the GA. During each generation, a set of new chromosomes, called an *offspring*, is formed by: (i) *selection* of a *mating pool*, i.e., a quota of parent chromosomes from the current population, according to the fitness values; (ii) combination of pairs of parents via the *crossover* genetic operator; (iii) modification of offspring chromosomes via the *mutation* genetic operator. The new chromosomes are then *decoded* in terms of domain solutions. Finally, a new generation is formed by *reinserting*, according to their fitness, some of the parents and offspring, and rejecting the remaining individuals so as to keep the population size constant.

The design of a GA for a given domain problem requires the specification of the following major elements: (i) a genetic coding of a solution; (ii) a choice of genetic operators and parameters; (iii) a fitness function, to evaluate a solution. These issues are covered in next section.

$$\begin{bmatrix} v_{1,1} \cdots v_{1,j} \cdots v_{1,n} \\ \vdots \\ \boxed{v_{i,1} \cdots v_{i,j} \cdots v_{i,n}} \\ \vdots \\ v_{l,1} \cdots v_{l,j} \cdots v_{l,n} \end{bmatrix} \leftarrow i\text{-th gene}$$

Figure 6.2. Genetic coding of a test pattern.

6.3 The Genetic Algorithm

GABES is a genetic algorithm-based environment aimed at producing a *test set* (TS), i.e., a set of test patterns, each one selected from the population generated at some step of a GA. More precisely, the GA maintains a *Dynamic Global Record Table* (DGRT) [147] containing a list of test patterns with the respective sets of detected faults. At each generation, the fitness of each individual from the population is evaluated. Then the individuals are examined in descending order of fitness and an individual is inserted in the DGRT if it detects faults that have not yet been found by previously inserted individuals. The construction of the DGRT is completed when its entries cover all faults (or a preset number of iterations has been reached), and the test patterns in the table are the final TS. The information in the DGRT is also used to compute the fitness function, as explained in Section 6.3.4.

A design choice in the development of the tool was to use a relatively lightweight GA, leaving a larger share of the computational burden to the already available simulator. Other proposals in the literature have different approaches, where the GA has access to details of the circuit structure and functionality, whereas the GA on which GABES is based relies only on the externally observable behavior of the simulated circuit.

6.3.1 The Genetic Coding

Single test patterns are considered as individuals (or chromosomes) in the GA. Their genetic coding, described below, is a matrix of logic values.

Let V_i be an input vector at clock cycle i, i.e., $V_i = [v_1, \ldots, v_n]$, where n is the number of input signals of the circuit, and the v's are the respective values. A test pattern (TP) is a sequence $[V_1, \ldots, V_l]$ of consecutive input vectors, where l is the number of clock cycles (or *length*) of the test pattern. Therefore, a test pattern is represented by a matrix of size $l \times n$, as shown in Figure 6.2.

The i-th row of the matrix represents the gene corresponding to the input vector V_i applied at the i-th clock cycle. The j-th column corresponds to the sequence of values on the j-th input pin.

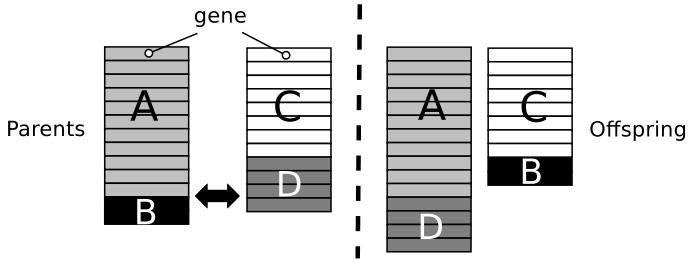


Figure 6.3. A scenario for the crossover operator.

It is worth noting that the number of genes in chromosomes is not assumed to be constant, since the number of clock cycles can take a different value for each test pattern.

6.3.2 The Genetic Operators and Parameters

Crossover is the main genetic operator. It consists in splitting two chromosomes in two or more sub-sequences and obtaining two new chromosomes by exchanging gene sub-sequences between the two original chromosomes. The place where a sub-sequence starts is called a *cut-point*. More specifically, we adopt a *single-point* crossover (Fig. 6.3) by choosing a non-uniform cut-point for each parent and generating the descendants by swapping the segments containing the ending clock cycles. The rationale for this choice is summarized in the following considerations.

With sequential logic, the output of a circuit depends on both the current input values and the previous inputs, starting from the initial state. Therefore, in order to take advantage of the added benefit of a gene sequence, in terms of number of recognized faults, we should take into account the state of the circuit, which is a result of all previous inputs, i.e., the previous gene sequence. Hence, it is generally more efficient to have a new generation chromosome retain a large fraction of the previous sequence.

In order to achieve this behaviour, we added the following criterion in the crossover operation: Random cut-points are generated via the probability density function of an exponential distribution, i.e., $f(x;\lambda) = \lambda e^{-\lambda x}$, where x is the distance of the cut point from the end of the sequence. This distribution implies that a large initial segment is kept unchanged from parent to child. Consequently, the end segments that are swapped are relatively short. The level of exploitation of the previous gene sequences can be adjusted via parameter λ.

The crossover operator is applied with a probability p_c (*crossover rate*) on the selected pair of individuals. When the operator is not applied, the offspring is a pair of identical copies, or *clones*, of the parents.

A higher crossover rate allows a better exploration of the space of solutions. However, too high a crossover rate causes unpromising regions of the search space to be explored. Typical values are in the order of 10^{-1} [81].

Mutation is an operator that produces a random alteration in a single bit of a gene. Mutation is randomly applied. The *mutation rate*, p_m, is defined as the probability that an arbitrary bit of an arbitrary gene is complemented. If it is too low, many genes that would have been useful are never discovered, but if it is too high, there will be much random perturbation, the offspring lose their resemblance to the parents, and the GA loses the efficiency in learning from the search history. Typical values of p_m are in the order of 10^{-2} [131]. We control the mutation operator by a dynamic p_m which is linearly decreasing between an initial value $\overline{p_m}$, and a value $\underline{p_m}$ at the final generation.

With a linearly decreasing p_m, the early generations have a high probability of mutation and solutions are spread all over the solutions space, so that most of them have a chance to be tried. Later generations have a lower mutation probability, so that the search is focused on the regions of the solution space where fitter individuals are found.

6.3.3 Selection Method

A selection operator chooses a subset of chromosomes from the current population. Various stochastic selection techniques are available. GABES uses the *roulette wheel* method [101]. With this method, an individual is selected with a probability that is directly proportional to its fitness. Each individual is mapped to an arc of a circle whose length equals the individual's fitness. The circumference is then equal to the sum of the fitnesses. Selection is made by choosing a random number with a uniform distribution between 0 and the circumference. The selected individual is the one mapped to the arc containing the chosen point. This ensures that better fit individuals have a greater probability of being selected, however all individuals have a chance.

6.3.4 The Fitness Function

The fitness function measures the quality of the solution, and is always problem dependent. In our approach, fitness takes into account the fault coverage achieved by each test pattern, its length, and its effectiveness in finding *hard* faults.

The fitness function adopted by GABES relies on a DGRT to evaluate each test pattern with respect to the performance of previously generated test patterns.

The fitness function of a test pattern i is defined in terms of a value c_i that we call the *relative efficiency* of the TP:

$$c_i = \sum_{j=1}^{n_i} (\xi_{ij} + 1)^{-k} \ ,$$

where n_i is the number of faults detected by the i-th test pattern; ξ_{ij} is the number of test patterns, generated before the i-th one, that detect fault j; k is is a configurable parameter of the algorithm, ranging in $[0, 1]$. The data required to compute ξ_{ij} are kept in the DGRT.

The fitness function is then

$$f(i) = \frac{c_i}{N} - M\frac{l_i}{L} ,$$

where N is the number of injected faults; M is a configurable parameter of the algorithm, ranging in $[0, 1]$, that represents the relative cost per clock cycle. For M equal to zero, the optimization process tends towards a maximum coverage. For increasing values of M, the fitness function penalizes also large test patterns. Parameter l_i is the length of the i-th test pattern; L is the maximum length of the test patterns. This parameter is heuristically chosen, depending on the size and complexity of the circuit.

Table 6.1 summarizes the parameters of the fitness function, together with those occurring in the TP generation algorithm (Sec. 6.3.5). The table also reports the values assigned to the parameters in the experiments discussed in Sect. 8.4.

The function increases with an individual's relative efficiency c_i. This value increases with the number of faults detected by individual i, but the weight of each detected fault j decreases with the number ξ_{ij} of other individuals that have been shown to detect the fault before i. The number ξ_{ij} is obtained from the DGRT and indicates how easily a fault can be detected (statistically, easy faults are detected earlier and more often). In this way, individuals that detect harder-to-find faults are rewarded. With higher values of parameter k, easy faults detected by a test pattern add a smaller contribution to its fitness.

6.3.5 Producing the test set

The final TS is obtained by an overall algorithm that iteratively evaluates a population of test patterns, inserts the best ones in the DGRT, and calls the GA proper to improve the population. The GA, in turn, uses the DGRT to compute the fitness function, as shown in the previous subsection. This is described more formally in Algorithm 4, where s identifies the iterations of the algorithm (up to a limit of $s_{\max}$ iterations), D is the DGRT, and P_s is the test pattern population at iteration s. The size of the population is S, and N is the number of possible faults. Parameter S is chosen so as to guarantee adequate diversity among individuals while limiting the computational cost of fitness evaluation. Predicate $\mathrm{improve}(m)$ is false when a *stall* condition occurs, i.e., when no improvement in the fitness of the best individual of each generation is achieved over the last m iterations.

For ease of notation, we assume that test patterns and faults are identified by natural numbers. The DGRT is represented as a set of pairs (i, j), such that test pattern i detects fault j. Function $\mathrm{coverage}(D)$ is the number of faults detected by the test patterns recorded in the DGRT, and $\mathrm{detects}(i, j)$ is true if and only if test pattern i detects fault j. The set of faults $\mathrm{detected}(i)$ found by test pattern i is updated in the course of the simulation. Predicate $\mathrm{finds}(i, \bar{n}, D)$ is true if and only if test pattern i detects a set of at least $\bar{n}$ faults not yet recorded in the DGRT, and $\mathrm{new}(i, \bar{n}, D)$ returns the pairs $(i, j_1), \ldots, (i, j_{\bar{n}})$ such that test pattern i detects a fault in that set.

The value of $\bar{n}$ is set initially at $0.05 \cdot N$ for smaller circuits and $0.1 \cdot N$ for larger ones, and is changed to 1 when the number of undetected faults drops below the initial value of $\bar{n}$.

The GA $\mathrm{ga}(P_{s-1}, D)$ produces the new generation P_s from the previous one, using the DGRT to compute the fitness. On exit from the outermost loop, function $\mathrm{compact}(D)$ produces a new DGRT by removing individuals whose faults are covered by other ones. Finally, (D) returns the test patterns contained in the DGRT.

The algorithm stops when one of the following conditions holds: (i) total fault coverage is achieved, or (ii) a stall condition is met, or (iii) the maximum allowed number of iterations $s_{\max}$ is reached.

$s \leftarrow 0$
$D \leftarrow \emptyset$
$D' \leftarrow \emptyset$
$P_0 \leftarrow (S \text{ randomly generated test patterns})$
while $\mathrm{coverage}(D) < N \wedge \mathrm{improve}(m) \wedge s < s_{\max}$ **do**
  for $i = 1$ **to** S **do**
    for $j = 1$ **to** N **do**
      if $\mathrm{detects}(i, j)$ **then**
        $\mathrm{detected}(i) \leftarrow \mathrm{detected}(i) \cup \{j\}$
      end if
    end for
    if $\mathrm{finds}(i, \bar{n}, D)$ **then**
      $D \leftarrow D \cup \mathrm{new}(i, \bar{n}, D)$
    end if
  end for
  $s \leftarrow s + 1$
  $P_s \leftarrow \mathrm{ga}(P_{s-1}, D)$
end while
$D' \leftarrow \mathrm{compact}(D)$
return (D')

Algorithm 4: The overall algorithm.

In Algorithm 5, P is the current population, P_M is the mating pool, A is the offspring, i.e., the set of individuals resulting from crossover and mutation, B is the set of individuals passed unchanged to the next generation, and Q is the size of the mating pool (Sec. 6.2).

Function $\mathrm{select}(P)$ returns an individual from P, selected with the roulette wheel method, and function $\mathrm{pair}(P_M)$ returns a pair of parents from P_M, selected with the roulette wheel method. Function $\mathrm{crossover}(x, y, \lambda)$ returns the offspring of a pair of parents, with λ as the *level of exploitation* parameter for cut point selection (Sec. 6.3.2). Mutation is then applied to the selected parents with probability p_m, and the mutated individuals are added to set A.

$P_M \leftarrow \emptyset$
$A \leftarrow \emptyset$
$B \leftarrow \emptyset$
for $i = 1$ **to** Q **do**
  $x \leftarrow \text{select}(P)$
  $P_M \leftarrow P_M \cup \{x\}$
end for
for $i = 1$ **to** $Q/2$ **do**
  $(x, y) \leftarrow \text{pair}(P_M)$
  $(x', y') \leftarrow \text{crossover}(x, y, \lambda)$
  $x'' \leftarrow \text{mutate}(x', p_m)$
  $y'' \leftarrow \text{mutate}(y', p_m)$
  $A \leftarrow A \cup \{x'', y''\}$
end for
for $i = 1$ **to** $S - Q$ **do**
  $x \leftarrow \text{select}(P, p_c)$
  $B \leftarrow B \cup \{x\}$
end for
$P \leftarrow A \cup B$
return P

Algorithm 5: The genetic algorithm.

Finally, a set B is built, with cardinality $S - Q$, with individuals drawn from the population P passed to the algorithm. The new generation is then obtained by replacing P by the union of A and B.

It may be observed that all sets used in the algorithm may contain pairs of identical individuals, due to the random character of the various operators. However, each individual is identifiable even when it is structurally identical to another one, therefore all sets are proper sets (not multisets). As a consequence, the cardinality of P is a constant.

6.4 The Test Pattern Generation Environment

The GA discussed above is coupled with the simulation-based fault injection tool for FPGAs presented in [34]. In this tool, the netlist of a digital circuit is modeled with the *Stochastic Activity Networks* (SAN) [170] formalism using the Möbius [63] modeling and analysis tool.

The test pattern generation process is shown in Fig. 6.4. In the figure, the block labelled “FPGA Design Process” is performed by an external tool that produces a netlist described in the EDIF language. This description is parsed and translated into the format used by the simulator to instantiate the model of the FPGA-based system, so the tool can seamlessly interact with the standard design process of an FPGA application.

Table 6.1. Parameters of GABES.

Parameter	Description	Value	Reference
λ	rate parameter for cutpoint distribution	1	Sec. 6.3.2
p_c	crossover rate	0.8	Sec. 6.3.2
$\overline{p_m}$	maximum mutation rate	0.15	Sec. 6.3.2
$\underline{p_m}$	minimum mutation rate	0.05	Sec. 6.3.2
N	number of possible faults	–	Sec. 6.3.4
M	penalty coefficient for TP length	0.5	Sec. 6.3.4
L	maximum length for TPs	10000	Sec. 6.3.4
k	penalty exponent for easy-to-detect faults	0.75	Sec. 6.3.4
$s_{\max}$	maximum number of iterations	2000	Sec. 6.3.5
S	TP population size	200	Sec. 6.3.5
m	number of iterations considered for stall condition	20	Sec. 6.3.5
$\bar{n}$	threshold for acceptance into DGRT	–	Sec. 6.3.5

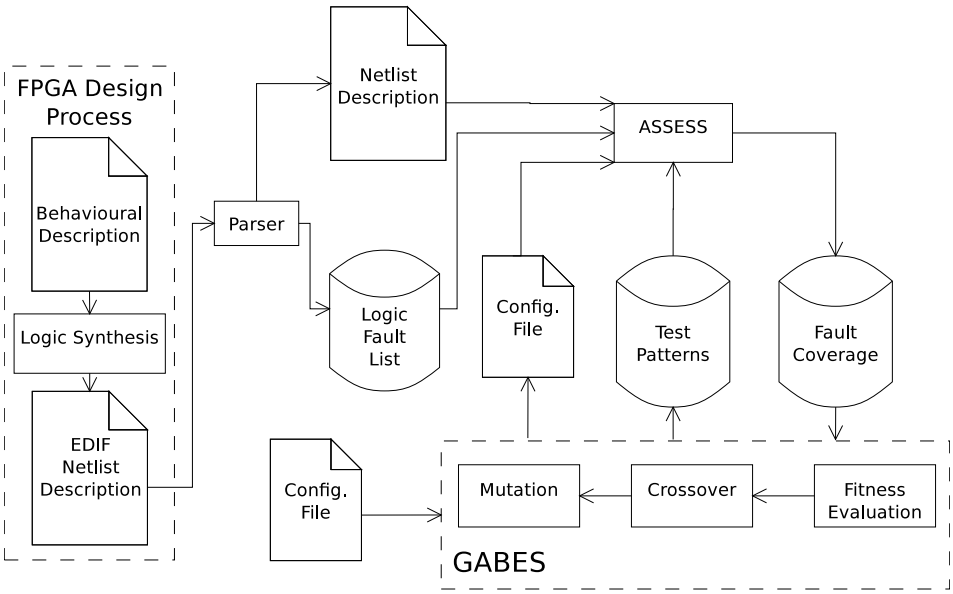


Figure 6.4. The Test Pattern Generation process.

The GA feeds the fault simulator with the current population of test patterns and then it waits for the fault coverage values produced as output of the simulations. These values are then used to update the DGRT and compute the fitness functions of the test patterns, leading to the next generation of the GA.

This GA is an efficient pattern generator thanks to the iterative processing of blocks of test patterns, which appreciably reduces the search space. It is worth noting that its genetic operators have the following properties: (i) At each generation, selection chooses test patterns which are better than average; (ii) crossover creates

groups of similar patterns to avoid worsening the quality of the selected patterns; and (iii) mutation creates dissimilar patterns without interfering with the result of crossover, especially in the later generations.

7

The SEU Analysis and Test Environment

All the tools previously presented work in conjunction with an EDIF parser and with the *E*^{2}*STAR* tool [37]. In particular the parser is a tool able to translate the EDIF description of the netlist into an intermediate description of the topology of the netlist in terms of connections among logic components and functionalities performed by components. This description is composed of a number of entries (one for each component in the netlist) of the form shown below:

$$i \;\; comp\_type_i \;\; comp\_fnct_i \;\; n\_inputs_i \;\; input\_comp_i^1 \;\; \ldots \;\; input\_comp_i^{n\_inputs_i}$$

where i represents the id of the component, $comp\_type_i$ represents the type of component i (LUT, flip-flop, multiplexer or I/O buffer), $comp\_fnct_i$ represents the function performed by component i (the truth table for LUTs, the type of flip-flop, input or output for buffers), n_inputs_i represents the number of input pins of component i and finally $input\_comp_i^j$ represents the id of the component whose output is connected with the j^{th} input pin of component i. Examples of entries are listed below:

```
2 ibuf 1;
3 lut 1100100100000000 4 1 15 0 17;
15 fdc 2 12 2;
```

In the examples component 2 is an input buffer connected to the input pin 1; component 3 is a LUT, implementing the truth table 1100100100000000 and connected with components 1, 15, 0 and 17; finally component 15 is a flip-flop with asynchronous reset, connected with component 12 (data signal) and with component 2 (clear signal).

Moreover the parser is able to produce a list of the effects of SEUs occurring in configuration bits associated with the logic resources used by the system under analysis. These faults are represented in terms of the induced modification of the truth table of the affected LUT.

E^2*STAR* is a static analyzer of the configuration memory of the SRAM-based FPGA device developed at the Politecnico di Torino. Given an FPGA device and a placed-and-routed design, *E*^{2}*STAR* is able to determine which are the configuration bits actually used by the design. Further *E*^{2}*STAR* is also able to determine which are the logical effects of SEUs occurring in the configuration bits controlling routing resources according to the fault model presented in Chapter 3. For each configuration bit associated with a routing component *E*^{2}*STAR* reports the number of propagation points of an SEU occurring in the configuration bit, and for each propagation point *E*^{2}*STAR* reports the logical effect, the affected component(s) and the pins of the affected component(s) to which the fault propagates. An example of entry is listed below:

```
12 3;
0 b 12 0 4 0;
1 wa 32 1 11 3;
2 sa0 23 2;
```

This example is related to an SEU having sequence number 12 and three propagation points (see the first line). In the first propagation point the SEU propagates as a bridge between pin 0 of component 12 and pin 0 of component 4. In the second propagation point the SEU propagates as a wired-and between pin 1 of component 32 and pin 3 of component 11. Finally in the third propagation point the SEU propagates as a stuck-at 0 on pin 2 of component 23.

The overall structure of the SEU analysis and test environment is shown in Figure 7.1. After the HDL specification of the system has been synthesized, the netlist description file and the list of the effects of SEUs in the configuration bits controlling logic resources are generated by the parser from the EDIF representation of the netlist. Starting from the synthesized netlist, the place-and-route algorithm performs the placement and routing of the designed system. After this step, *E*^{2}*STAR* can be performed on the post-place-and-route netlist description.

More in detail, the use of the proposed software tools involves the following steps:

1. Producing the Verilog/VHDL behavioural description of a system;
2. Obtaining the EDIF description of the netlist of the system with the support of a synthesis tool;
3. Translating the EDIF netlist into the intermediate language using the EDIF parser, obtaining the intermediate description of the netlist and the list of the effects of the SEUs in the configuration bits controlling logic resources;
4. Placing-and-routing the netlist on a target device using a place-and-route tool;
5. Running *E*^{2}*STAR* on the placed-and-routed netlist obtaining the list of the effects of SEUs in the configuration bits controlling routing resources;
6. For the execution of ASSESS the following additional steps have to be executed:

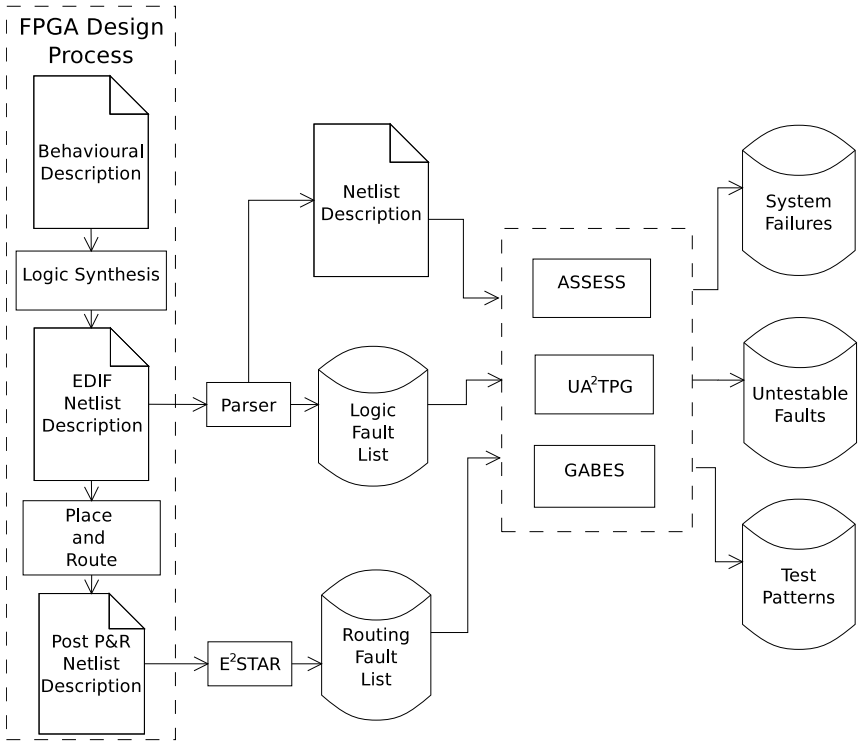


Figure 7.1. Flow Diagram of the SEU Analysis and Test Environment.

a) Configuring the simulator specifying the number of simulated clock cycles, the type of simulation, and the required outputs; and finally
b) Executing ASSESS.

7. For the execution of UA^{2}TPG the following additional steps have to be executed:
   a) Executing UA^{2}TPG.
   b) Compressing the set of test patterns generated by UA^{2}TPG, by eliminating duplicated test patterns and those test patterns that are prefix of longer test patterns
8. For the execution of GABES the following additional steps have to be executed:
   a) Configuring the genetic algorithm specifying the parameters of the GA and the characteristics of the required solutions; and finally
   b) Executing GABES.

It may be observed that the proposed environment is fully integrated in the standard design process of FPGA-based systems. In particular, as previously noted, all the proposed tools take an intermediate representation of the netlist produced by a parser that takes as input the EDIF file produced by the synthesis tool. Moreover, the *E^{2}STAR* works directly on the post place-and-route netlist description file.

8

Experimental Results

8.1 The considered circuits

In order to test the capabilities of the proposed tools we applied it to some circuits from the ITC'99 benchmark [56]. The tested benchmark circuits provide a diversified set of test cases composed of sequential circuits with a single clock signal, no tristate buses or internal memories, modeled at the RTL level, ranging from 6 to $8,000$ equivalent gates and from 4 to 59 FFs. We synthesised the VHDL code of the circuits using the Xilinx ISE CAD tool. As a target device we adopted the Xilinx Virtex-II XC2VP30 device. The characteristics of the designs used in the experiments are shown in Table 8.1, which reports for each circuit the number of SEUs affecting logic and routing resources (column L-SEUs and R-SEUs respectively), the number of Look-Up Tables (LUTs), Flip-FLop (FFs), MUXes, Input and Output buffers. Table 8.2 shows the functionality implemented by each circuit. The computer used for the experiments was equipped with an Intel Core i5 (QuadCore) 2.67 GHz, 256 KB L1 Cache, 1 MB L2 Cache, 8MB L3 Cache, 4 GB RAM.

In Table 8.3 we show the results of the analysis performed with *E^2STAR* to the considered circuits. The table shows the number of critical configuration memory bits (RoutingFaults) identified by the tool, and the number of affected nodes classified by logical effect: Stuck-at-0, Stuck-at-1, Wired-And, Wired-Mix, and Bridge. Wired-Or effects were not observed. It may be observed that, as we previously discussed, the number of propagation points per SEU in the configuration bits controlling the routing structure is much higher than the actual number of SEUs itself. In particular the number of fault propagation points is on average 5.3 times larger than the number of faults (6.06 times larger for the b07 circuit). Moreover, as it was discussed in Chapter 3, the largest number of effects of SEUs is stuck-at 0 and stuck-at 1. This, together with the accurate model of the electrical effects induced by SEUs in PIPs, makes the fault model implemented by the simulator much accurate than the classical open/short fault model for circuit interconnections.

Table 8.1. Characteristics of the considered benchmark circuits.

Circuit	L-SEUs	R-SEUs	LUTs	FFs	MUXs	IBuffs	OBuffs
b01	124	547	9	5	0	3	2
b02	52	304	4	4	0	2	1
b03	954	5,910	76	37	0	5	4
b06	104	566	9	8	0	3	6
b07	1,720	10,431	152	51	20	2	8
b08	504	2,689	40	21	0	10	4
b09	692	3,872	53	28	0	2	1
b10	660	3,942	52	24	0	12	6
b11	1,776	10,104	147	38	14	8	6
b13	1,216	7,203	106	59	11	11	10

Table 8.2. Benchmark circuit functions.

Circuit	Function
b01	Compare serial flows
b02	Recognize binary coded decimal numbers
b03	Resource arbiter
b06	Interrupt handler
b07	Count points on a straight line
b08	Find inclusions in sequences of numbers
b09	Serial-to-serial converter
b10	Voting system
b11	Scramble string with variable cipher
b13	Interface to meteo sensors

8.2 Results from the application of ASSESS

Each circuit was simulated by applying $10,000$ randomly generated test patterns and performing an exhaustive fault injection. For each circuit, the same test vectors and faults were also applied to and injected into its prototype on the fault injection board. This experiment allowed us to validate the proposed fault simulator.

The comparison between the two sets of results (fault simulation and fault injection) is shown in Table 8.4. The table shows in the second column (TotFaults) the total number of faults (both in logic and routing elements), in the third column (*Sim-DF*) the number of faults that caused a failure of the systems during the SEU simulation, in the fourth column (*Sim-T*) the simulation time (in minutes), in the fifth column (*FI-DF*) the number of faults that caused a failure of the system during the fault injection

Table 8.3. Effects of SEUs in the routing elements.

Circuit	RoutingFaults	SA0	SA1	W-AND	W-Mix	Bridge
b01	547	708	2,944	5	7	0
b02	304	118	339	5	7	102
b03	5,910	8,105	21,661	1,423	1,431	2,320
b06	566	372	790	0	18	305
b07	10,431	18,331	47,762	4,085	3,911	4,739
b08	2,689	3,074	8,061	464	496	1,217
b09	3,872	6,569	15,948	567	512	1,908
b10	3,942	4,603	10,727	482	692	1,498
b11	10,104	14,059	35,749	3,537	3,536	4,480
b13	7,203	10,390	27,720	1,143	1,387	3,602

Table 8.4. Results from SEU simulation and fault injection

Circuit	TotFaults	Sim-DF	Sim-T(min)	FI-DF	FI-T(min)
b01	676	676	0.45	676	60.84
b02	359	352	0.11	350	32.31
b03	6,873	3,285	1,470.90	3,278	518.57
b06	679	670	0.56	670	61.11
b07	12,161	927	15,709.98	927	1,094.49
b08	3,207	157	189.73	157	288.63
b09	4,567	2,081	308.26	2,080	411.03
b10	4,620	3,548	365.75	3,545	415.80
b11	11,894	7,521	8,587.46	7,519	1,070.46
b13	8,440	2,517	2,424.14	2,515	756.6

experiment, and in the sixth column (*FI-T*) the time (in minutes) used by the fault injector.

The comparison between columns Sim-DF and FI-DF in Table 8.4 shows that the proposed simulation method is able to accurately reproduce the effects of SEUs affecting any configuration bit of an SRAM-based FPGA system. In particular the comparison with results obtained by fault injection show that our simulator has an average error of 0.1%, with a maximum error for b02 of 0.5%.

The accuracy of the proposed simulator is even more evident if we look at Table 8.5, where the results (in terms of SEU sensitivity) obtained with the proposed simulator are compared with the results obtained with the same simulator but considering stuck-at faults (that is what commercial and academic fault simulators are today able to do) instead of the accurate fault model previously discussed and with

Table 8.5. Estimated SEU Sensitivities Comparison

Circuit	SimSens	Sim^{SA}Sens	FISens
b01	100.0%	100.0%	100%
b02	98.0%	100.0%	97.5%
b03	47.8%	45.9%	47.7%
b06	98.7%	100.0%	98.7%
b07	7.6%	5.1%	7.6%
b08	4.9%	2.1%	4.9%
b09	45.5%	37.6%	45.5%
b10	76.8%	76.2%	76.7%
b11	63.2%	80.5%	63.2%
b13	29.8%	27.1%	29.7%

results obtained by fault injection. As we can see from Table 8.5 the proposed SEU simulator (SimSens column) and the fault injection experiments (FISens column) estimate almost the same SEU sensitivity, while using a stuck-at based fault simulator (Sim^{SA}Sens column) we obtain very different results. This is due to two reasons: (i) the number of stuck-at faults is much smaller than the actual number of SEUs that may occur in the configuration memory of an SRAM-based FPGA system; and (ii) as we previously discussed, the activation and propagation for stuck-at faults is completely different than for SEUs affecting the configuration memory controlling logic components on the one hand, and controlling routing resources on the other hand.

In particular, while, as we previously discussed, the error of ASSESS with respect of the fault injection is 0.1% on average, with a maximum 0.5%, the stuck-at fault simulation has an average error of 15.1% and a maximum error, for the b08 circuit, of 56.2% with respect of the fault simulation. This experiment clearly shows that the analysis of the sensitivity to SEUs in the configuration memory performed by ASSESS is extremely more accurate than the one performed by the existing simulators of faults that consider the stuck-at fault model to emulate the behaviour of a faulty component in the netlist.

Using ASSESS the average time per simulated fault is dependent on the size and complexity of the circuit. Even if for small circuits the simulator is faster than the fault injector, on average, as expected, the simulator is slower than the fault injector. The time needed to simulate large circuits is much longer the the time required by fault injection. Nevertheless the proposed tool could be used early during the design process, thus allowing designers to assess early the robustness to SEUs of the circuit. In this way ASSESS could bring two benefits to designers: (i) system modifications and corrections due to discovered weaknesses against SEUs could be performed early, thus allowing designers to save money and time; and (ii) the final radiation testing and fault injection experiments can be performed on a prototype of the system

Table 8.6. Simulation Time Comparison

Circuit	Sim-T(min)	Sim^{SA}-T(min)	FI-T(min)
b01	0.45	0.04	60.84
b02	0.11	0.01	32.31
b03	1,470.90	119.13	518.57
b06	0.56	0.04	61.11
b07	15,709.98	1798.58	1,094.49
b08	189.73	27.95	288.63
b09	308.26	30.18	411.03
b10	365.75	24.13	415.80
b11	8,587.46	697.27	1,070.46
b13	2,424.14	523.57	756.6

that has been already hardened against SEUs, thus, again, allowing designers to save money and time. Moreover, as we have previously shown, classical stuck-at fault simulators are not able to accurately reproduce the behaviour of the faulty system, even if they are much faster than ASSESS (as shown in Table 8.6), while ASSESS is able to estimate the SEU sensitivity of the system with a very low error. If we consider these points, we believe that the long time needed by the proposed SEU simulator does not represent such a high cost.

In order to show the usage of ASSESS for failure probability analysis we performed the following experiment: we considered the SEU rate of 82 s^{-1} reported in [7]; thus assuming a working frequency of 1MHz, we had a SEU rate of $8{,}2 \cdot 10^{-8}$ per clock cycle. We point out that this is just a usage example with some realistic values for SEU occurrence probability and working frequency of the system. We simulated the circuits b08, b09 and b10 for 500,000 up to 5,000,000 clock cycles (thus simulating 0.5 up to 5 seconds) randomly injecting SEUs during the simulation. For each time duration we performed 1,000 up to 5,000 simulation runs in order to obtain failure probability values with a confidence level of 0.95 and a confidence interval of 0.1. Results from this experiment are shown by Figure 8.1 and by Figure 8.2.

Figure 8.1 represents the single-SEU failure probability of the tree circuits: only one SEU per simulation run was randomly injected in the circuit, thus allowing us to assess the failure probability due to single SEUs.

Figure 8.2 represents the accumulation-SEU failure probability of the tree circuits: N_{cb} SEUs per simulation run were randomly injected in the circuit, being N_{cb} the number of configuration bits actually used by the system, thus allowing us to assess the failure probability due to SEU accumulation.

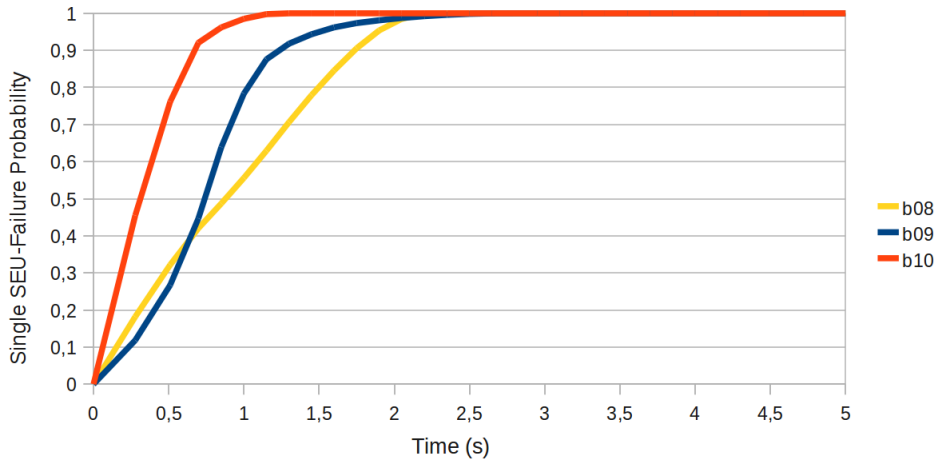


Figure 8.1. Failure probability with single SEU injection.

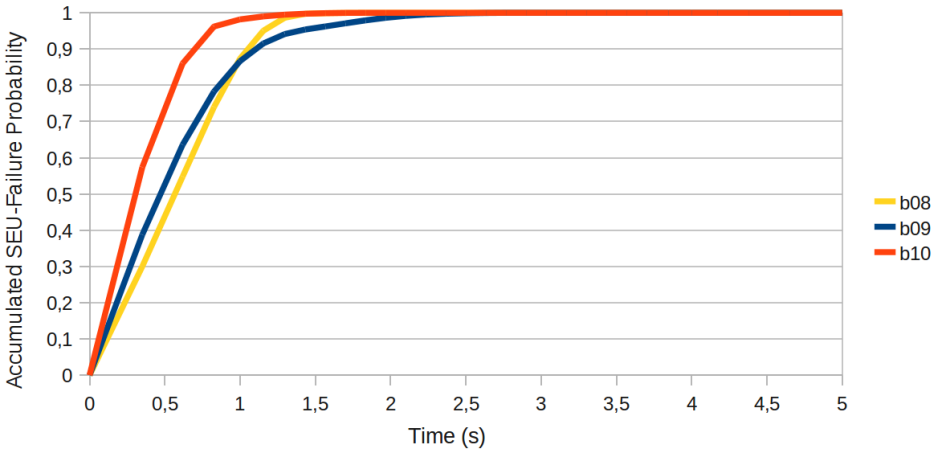


Figure 8.2. Failure probability with multiple SEU injection.

8.3 Results from the application of UA^{2}TPG

In order to perform a sanity check of the proposed tool we tested some ITC'99 circuits with 100000 randomly generated test vectors and with the test patterns produced by the genetic algorithm presented in [30]. The result of this experiment was that neither testing technique was able to cover the faults that the proposed tool proved to be untestable. Further we performed the unexcitability analysis using the tools presented in [33] and in [36] on the circuits considered for validation. Results from this analysis showed that the set of untestable faults identified by the tool proposed in this paper always contained the set of unexcitable faults identified by the tools presented in [33] and in [36]. Finally, using the SEU simulator presented in [37], we checked that the test patterns generated by UA^{2}TPG were actually able to detect those faults that had

Table 8.7. Results from the application of UA^{2}TPG.

Circuit	TotF	UntLogF	UntRoutF	TotUntF	UntTime(min)
b01	671	0	0	0	1.29
b02	356	6	2	8	0.66
b03	6,864	694	214	908	62.8
b06	670	8	6	14	1.24
b08	3,903	76	66	142	16.34
b09	4,564	244	73	317	35.48
b10	4,602	288	136	424	31.39
b13	8,419	640	671	1,311	263.10

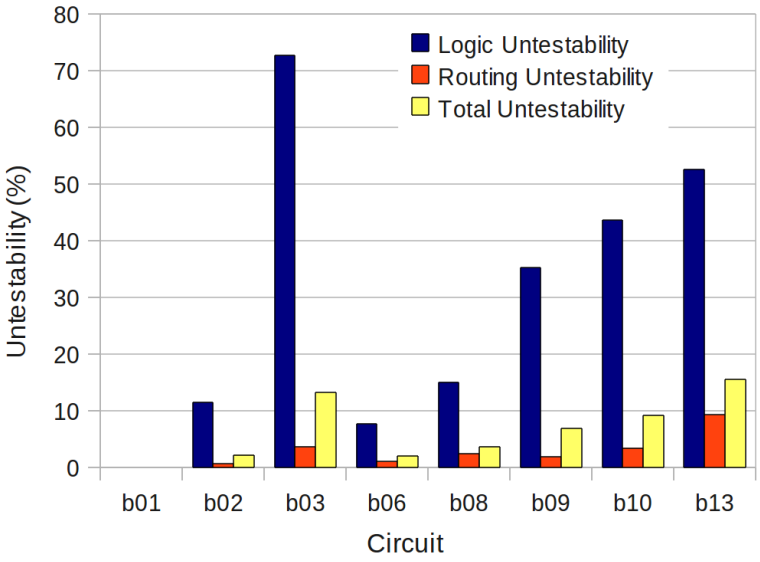


Figure 8.3. Fault untestability for the considered circuits.

been demonstrated to be testable. All these results reinforced our confidence in the correctness of the analysis.

Results obtained from the application of the proposed tool to the considered circuits are shown in Table 8.7. The table shows the circuit name, the total number of faults in configuration bits controlling both logic and routing resources (TotF), the number of untestable SEUs affecting logic resources (UntLogF column), the number of untestable SEUs affecting routing resources (UntRoutF column), the total number of untestable SEUs (TotUntF column) and the time (in minutes) needed by the tool to carry out the analysis (UntTime column). Figure 8.3 shows the percentage of untestable SEUs in configuration bits controlling logic and routing resources, and the total untestability percentage for each circuit.

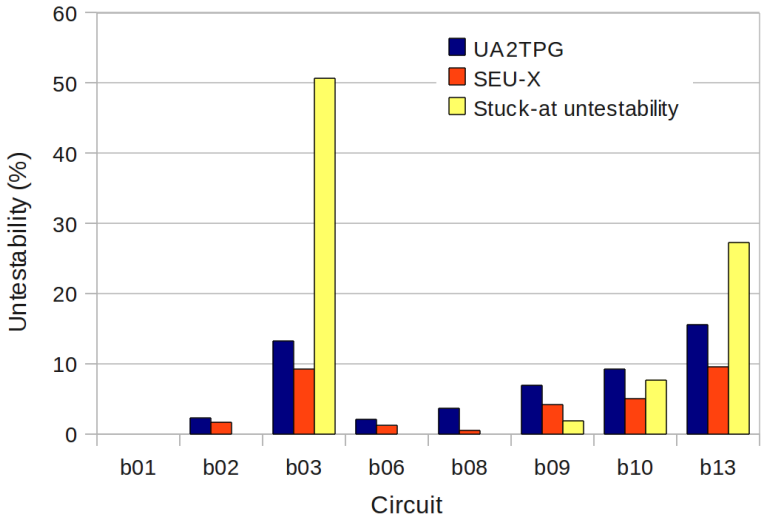


Figure 8.4. Fault untestability for the considered circuits obtained with UA^{2}TPG, with the tools proposed in [33] and in [36] and considering stuck-at faults.

The experiments show that almost all the considered circuits, except for b01, have a number of faults that cannot be tested. The average untestability is 6.6%. The highest untestability is 15.5% for b13, while the lowest is 2% for b06. SEUs in logic resources seem to be much harder to test than SEUs in routing resources. This may be explained, if we take two points into account: (i) the excitation of an SEU in a configuration bit controlling an LUT depends on the values of all the inputs of the LUT while, as we previously discussed, the excitation of an SEU in a configuration bit controlling a PIP depends on the value of one or two signals; and (ii) as we previously discussed, each SEU in the routing structure has a very large number of propagation points. If we consider only SEUs in logic resources we find an average untestability of 29.8%, with a peak of about 72.7% for b03. Considering only SEUs in routing resources we find an average untestability of 2.8%, with a peek of about 9.3% for b13.

The accuracy of the proposed tool is evident if we look at Figure 8.4, where the results (in terms of SEU untestability) obtained with the proposed tool are compared with the results obtained combining the tools for the SEU unexcitability analysis presented in [33] and [36] (SEU-X columns) and with a version of UA^{2}TPG modified in order to consider stuck-at faults (that is what similar commercial and academic tools are today able to do) instead of the accurate fault model previously discussed. As we can see from the figure, the combination of the tools presented in [33] and [36] (orange line) calculates a fault untestability that is always smaller than the untestability calculated by UA^{2}TPG. This is obvious if we consider that UA^{2}TPG addresses the whole untestability problem (fault activation and propagation) while in [33] and [36] only the problem of fault activation was considered. By comparing the analysis performed by UA^{2}TPG, with the stuck-at untestability analysis (light yellow line) we can see how

Table 8.8. Automatic Test Pattern Generation Results using UA^{2}TPG.

Circuit	TotF	TPLen
b01	671	538
b02	356	160
b03	6,864	1,111
b06	670	255
b08	3,903	89
b09	4,564	5,866
b10	4,602	5,614
b13	8,419	5,003

different the results are, and thus how the inaccuracy of the stuck-at fault model affects the analysis of the testability of faults, when SEUs in the configuration memory of SRAM-based FPGAs are considered. This is due to two reasons: (i) the number of stuck-at faults is much smaller than the actual number of SEUs that may occur in the configuration memory of an SRAM-based FPGA system; and (ii) as we previously discussed, the activation and propagation for stuck-at faults is completely different than for SEUs affecting the configuration memory controlling logic components on the one hand, and controlling routing resources on the other hand.

Table 8.8 shows the number of faults of each circuit (TotF column) and the length of the test patterns generated by the proposed tool (TPLen column). We point out that these test patterns are able to cover the 100% of the testable SEUs.

In order to show the accuracy of the test patterns generated by UA^{2}TPG for the accurate model of SEUs, with respect to test patterns generated for stuck-at faults we used the SEU simulator presented in [37] to evaluate the fault coverage obtained using the two sets of test patterns. As expected, test patterns generated by UA^{2}TPG for the accurate SEU model detected the 100% of the testable faults. Test patterns generated for the stuck-at fault model obtained much lower fault coverage values: 78.9% on average, maximum 93.16% for b02 and minimum 54.03% for b03. Results from this experiment are reported in Table 8.9 that for each circuit shows the fault coverage obtained with the test patterns generated for the accurate model of SEUs (column F-Coverage) and for stuck-at faults (column F-Coverage^{SA})

The time required for the analysis ranges from some seconds up to some minutes for very small and medium size circuits. For larger circuits the required time is about some hours. We believe that these times are reasonable if we take into account two different points: on the one hand this analysis should be performed just once during the design of the system; on the other hand, by automatically generating test patterns able to test the 100% of the testable SEUs, the proposed tool could substitute other ATPG tools used in the FPGA design process, thus producing an overall benefit for the design of the system.

Table 8.9. Comparison of fault coverages obtained with test patterns generated for SEU and for stuck-at faults.

Circuit	F-Coverage	F-Coverage^{SA}
b01	100%	92.89%
b02	100%	93.16%
b03	100%	54.03%
b06	100%	92.71%
b08	100%	61.33%
b09	100%	82.23%
b10	100%	73.54%
b13	100%	81.38%

8.4 Results from the application of GABES

The GABES test pattern generator has been applied to some circuits from the ITC'99 suite [56]. The VHDL code of the circuits was synthesized for the Virtex 4 target device using the Xilinx ISE tool [2]. The characteristics of the netlists, in terms of the number of LUTs, flip-flops (FFs), multiplexers (MUXs) and input and output buffers (IBuffs and OBuffs), are summarized in Table 8.10. The function of the circuits, as reported in Corno *et al.* [56], is shown in Table 8.2. The values of the parameters for the experiments are shown in Table 6.1.

Table 8.10. Characteristics of the circuits to which GABES has been applied.

Circuit	LUTs	FFs	MUXs	IBufs	OBufs
b01	15	10	1	3	2
b02	4	4	0	2	1
b03	90	35	1	5	4
b06	9	8	0	3	6
b08	47	21	1	10	4
b09	47	29	2	2	1
b10	55	24	0	12	6

In all experiments, only the excitable faults were injected. The unexcitability analysis of SEUs in the configuration memory was carried out with SEU-X [33].

The GA uses an adaptive DGRT admittance threshold policy, with DGRT compaction. The GA terminates if there is no improvement in the best fitness of the population over a predetermined number of generations, or when the preset maximum number of generations is reached.

To show the behaviour of the optimization process performed by the GA, in Figure 8.5 we report, for the ITC'99 b09 circuit, the fault coverage of the whole DGRT

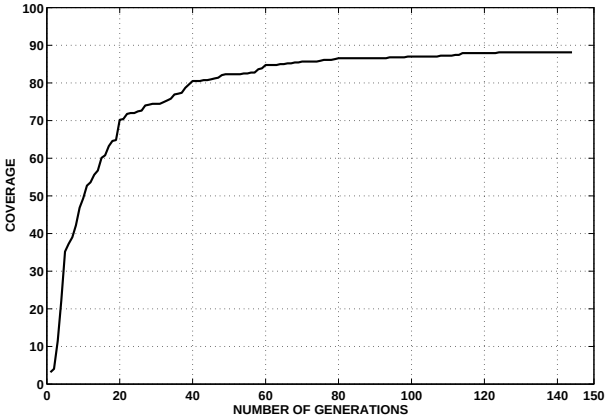


Figure 8.5. DGRT coverage vs. number of generations for b09.

(i.e., the cardinality of the union of the faults detected by each DGRT entry) at each generation versus the number of generations. Here the optimization process has been tuned so as to maximize coverage, at the cost of greater test length. The figure shows how a high coverage is achieved after a small number of generations.

For the same trial considered in Figure 8.5, Figure 8.6 represents the number of individuals in DGRT that detect a fault, for each generation and for each fault, before DGRT compaction. The number of individuals is represented via a grey level. Horizontal solid black lines show undetected faults, whereas light horizontal lines represent easy-to-detect faults. A number of solid black lines is expected, due to the presence of non-observable faults.

In Figure 8.7, the final status of the DGRT in terms of fault coverage is shown: for each individual in DGRT and for each fault, a white (black) dot represents a covered (uncovered) fault, respectively. Black horizontal lines represent undetected faults. It can be observed that many individuals are very similar in terms of detected faults. It has been experienced that this phenomenon occurs particularly when the fitness function is tuned to maximize coverage by loosening constraints on test length (choosing low or null values of M) and on the threshold for acceptance in the DGRT (choosing low values of $\bar{n}$). New individuals added in the DGRT may also detects faults already detected by other individuals, thus making the latter redundant. As an example, in the reported trial the compacting process reduced the length of the test set by 40.3%.

Results from the application of GABES for the generation of test patterns for the logic resources of the considered circuits are shown in Table 8.11. For each circuit, the *Faults* column reports the number of possible faults while the *Unex* column reports the number of unexcitable faults calculated with the SEU-X tool. The remaining columns (F_e, C_t, C_c, *Length*, and *Time*) report the percentage of excitable faults, the measured coverage with respect to all faults, the measured coverage with respect to excitable

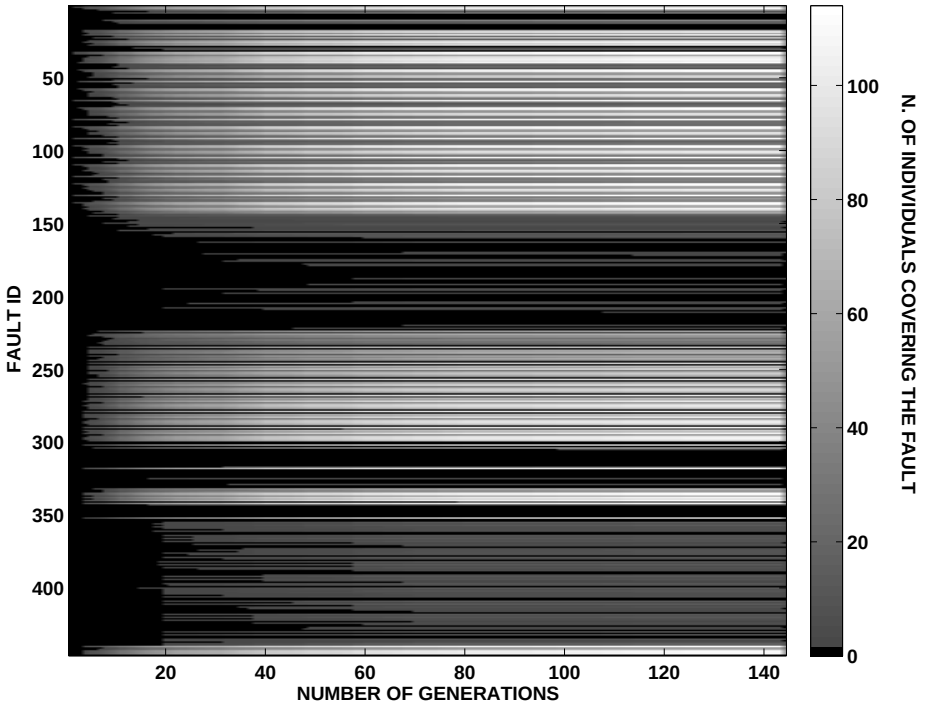


Figure 8.6. Detection of faults by GA generation.

Table 8.11. Experimental results using GABES.

Circuit	Faults	Unex	F_e (%)	C_t (%)	C_c (%)	Length	Time
b01	197	55	72.1	69.53	96.47	151	10.25
b02	55	6	89.1	87.26	97.95	163	1.32
b03	1083	508	53.1	43.11	81.21	2380	8620.29
b06	113	8	92.9	92.02	99.04	76	3.32
b08	590	47	92.0	79.99	86.92	6988	9630.87
b09	651	205	68.5	60.37	88.12	3033	2042.98
b10	726	208	71.3	71.3	100.0	1638	84.31

faults, the test length (cumulative number of clock cycles of the test set), and the simulation time (in minutes), respectively.

Table 8.12 compares the results for the multiple test patterns solution generated by the GABES tool (*Multiple TP GA* column) with those obtained by the same tool in *single test pattern* mode (*Single TP GA* column), and with those obtained by random testing (*Random* and *Random** columns).

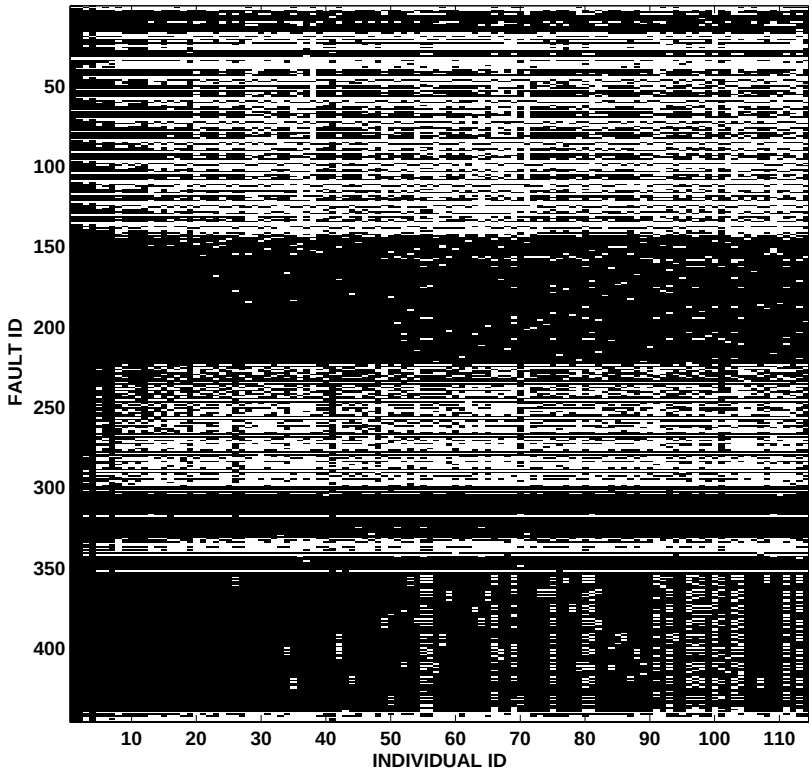


Figure 8.7. Final DGRT coverage.

The single test mode is a particular case of the TP generation algorithm, where the test set reduces to a single TP. In this mode, the DGRT is disabled and parameter k of the fitness function is set to zero. Two different random testing trials were performed: In the first one (*Random* column) a random test pattern of fixed length $(10$ thousand clock cycles) was used; in the second one (*Random** column) a random test pattern with the same length as the one produced in multiple test pattern mode was used.

Results from the application of the single test pattern mode to many of the considered circuits are missing because, given the very long execution time, it was unfeasible to apply the tool to circuits larger than b01, b02 and b06.

The multiple test pattern mode of the genetic algorithm made it possible to tackle larger circuits than the single test pattern mode.

It can then be argued that generating a test set instead of a single test pattern considerably improves the efficiency of the TP generation process, because each test pattern is independent with respect to the other ones in the test set.

Table 8.12. Comparison of the results obtained by GABES with other results.

Circuit	Multiple TP GA		Single TP GA		Random	Random*
	C_t (%)	Length	C_t (%)	Length	C_t (%)	C_t (%)
b01	69.53	151	64.97	120	69.0	37.05
b02	87.26	163	89.1	41	85.45	16.36
b03	43.11	2380	–	-	27.52	23.26
b06	92.02	76	92.9	72	92.0	50.44
b08	79.99	6988	–	-	2.7	2.7
b09	60.37	3033	–	-	6.3	5.83
b10	71.30	1638	–	-	38.56	38.7

Further, it may be observed that results obtained by the GA are much better than the ones obtained by random testing, either in terms of fault coverage or of test length, and for some circuits in terms of both. In particular, compared to the results in the *Random** column, the multiple test pattern solution achieves much higher fault coverages.

Comparing the values of C_t in the *Multiple TP GA* column in Table 8.12 with the *Random** column, an average improvement of 7.54 is observed, with a maximum improvement of 29.72 for the b09 circuit. With respect to the *Random* column, the solution generated by the genetic algorithm has always a shorter length, while the fault coverage is higher for large circuits and equals for the small ones. This reveals the better scalability of the Multiple TPGA with respect to the other approaches. For example, the solution generated by the Multiple TP GA for b09 is 3.03 times shorter and has a 9.58 times higher fault coverage than the Random solution (whose length is fixed to 10 thousand clock cycles).

Finally, we observe that fault coverages reported in the literature for other test pattern generators [56, 163], are generally much higher than the ones shown here. Those test pattern generators consider the stuck-at fault model, whereas the one discussed here addresses SEUs in any configuration bit. These faults are arguably more difficult to detect than stuck-at faults, and, as observed in Chapter 3, much more numerous. A comparison of our results with those in the current literature should take these differences into account.

9

Conclusions

In this dissertation a framework of tools for the analysis and test of the effects of SEUs occurring in the configuration memory of SRAM-based FPGA systems has been presented. All the proposed tools implement an accurate model of SEUs affecting the configuration bits controlling both logic and routing resources of the system. This makes the proposed tools much more accurate than similar commercial and academic tools currently available. The advantage of the proposed tools is that they can be applied early during the design process of the system, thus allowing designers to reach the final fault injection and radiation experiments with a prototype of a well studied and analysed system.

The framework is composed of ASSESS, a simulator of SEUs working on the netlist representation of the system; UA^{2}TPG, an untestability analyzer and automatic test pattern generator and GABES, a genetic algorithm-based tool for the generation and optimization of test patterns for in-service testing. We point out that these are the first software tools for the simulation, the untestability analysis and the test pattern generation specifically addressing SEUs in the configuration memory of SRAM-based FPGA systems.

The comparison between results obtained using ASSESS and by fault injection has shown that the proposed SEU simulator is able to very accurately reproduce the behaviour of a faulty FPGA-based system. Moreover, by comparing results obtained using ASSESS with results obtained by simulating stuck-at faults, we have shown that the proposed simulator is much more accurate than similar tools today available, even if the time required for the analysis performed by ASSESS is much longer than that required by other simulators. In particular the comparison between results obtained using ASSESS with those obtained by fault injection has shown that the proposed fault simulator has an average error of 0.1% and a maximum error of 0.5%, while using a stuck-at fault simulator the average error with respect of the fault injection experiment has been 15.1% with a maximum error of 56.2%. Finally we have shown how the proposed SEU simulator can be used to evaluate the effects of the accumulation of SEUs in the configuration memory of SRAM-based FPGA systems.

The application of UA^{2}TPG to some circuits from ITC'99 benchmarks has shown that the tool is able to identify the untestable SEUs much more accurately than other tools for the analysis of the testability of SEUs in the configuration memory of SRAM-based FPGAs and than other tools for the analysis of the testability of stuck-at faults in digital circuits. In particular by comparing the untestability results obtained using UA^{2}TPG for the accurate SEU model, with the results of the untestability analysis for stuck-at faults we find an average difference of 7.9% with a maximum of 37.4%. UA^{2}TPG is also able to generate test patterns to test the 100% of the testable SEUs in a reasonable time. Finally the comparison between fault coverages obtained by test patterns generated for the accurate model of SEUs and the fault coverages obtained by test pattern designed for stuck-at faults, shows that the former detect the 100% of the testable faults, while the latter reach an average fault coverage of 78.9%, with a minimum of 54% and a maximum of 93.16%.

The application of GABES to some circuits from the ITC'99 benchmark has shown that the adopted multiple test pattern genetic algorithm reaches good scalability and efficiency in terms of both fault coverage and length of the test patterns.

References

1. Guideline for FPGA IV&V Demonstration Program, January 2007. National Aeronautics and Space Administration (NASA).
2. ISE design suite software manuals and help. `http://www.xilinx.com/support/documentation/sw_manuals`, 2010.
3. M. Abramovici, J. J. Kulikowski, P. R. Menon, and D. T. Miller. Smart and fast: Test generation for vlsi scan-design circuits. *IEEE Des. Test*, 3(4):43–54, July 1986.
4. Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. John Wiley & Sons., 1990.
5. Miguel Angel Aguirre, Jonathan Noel Tombs, Vicente Baena, Fernando Muñoz, Antonio Jesus Torralba, A. Fernández-León, and F. Tortosa-López. Ft-Unshades: a New System for Seu Injection, Analysis and Diagnostics Over Post Synthesis Netlist. In *Proceedings of the 8th Military and Aerospace Programmable Logic Devices International Conference (MAPLD'05)*, 2005.
6. A.A. Al-Yamani, S. Mitra, and E.J. McCluskey. Optimized reseeding by seed ordering and encoding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(2):264 – 270, feb. 2005.
7. M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, A. Marmo, S. Pastore, and G. R. Sechi. A Tool for Injecting SEU-Like Faults into the Configuration Control Mechanism of Xilinx Virtex FPGAs. In *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '03)*, page 71, Washington, DC, USA, 2003. IEEE Computer Society.
8. M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, S. Pastore, and G.R. Sechi. Evaluation of Single Event Upset Mitigation Schemes for SRAM based FPGAs using the FLIPPER Fault Injection Platform. In *Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT '07)*, pages 105 –113, sept. 2007.
9. Altera Corporation. *Stratix Device Handbook*, January 2005.
10. Altera Corporation. *Stratix GX Device Handbook*, March 2005.
11. Altera Corporation. *Cyclone II Device Handbook*, February 2007.
12. Altera Corporation. *Stratix II Device Handbook*, May 2007.
13. Altera Corporation. *Stratix II GX Device Handbook*, August 2007.
14. Altera Corporation. *Cyclone Device Handbook*, May 2008.
15. Altera Corporation. *Cyclone III Device Handbook*, December 2009.
16. Altera Corporation. *Stratix III Device Handbook*, July 2009.
17. Altera Corporation. *Stratix IV Device Handbook*, November 2009.
18. Altera Corporation. *Arria II GX Device Handbook*, February 2010.
19. Altera Corporation. *Arria V Device Handbook*, December 2012.

20. Altera Corporation. *Cyclone V Device Handbook*, October 2012.
21. Altera Corporation. *Cyclone V Device Handbook*, December 2012.
22. Altera Corporation. *Stratix V Device Handbook*, December 2012.
23. Rossnyev Alvarado and David Herrell. Approach to Designing FPGA-Based Digital I&C Systems for Nuclear Applications. In *Proceedings of the 6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC&HMIT 2009)*. American Nuclear Society, April 2009.
24. G. Asadi and M. B. Tahoori. An Analytical Approach for Soft Error Rate Estimation of SRAM-based FPGAs. In *Proceedings of the 7th Military and Aerospace Programmable Logic Devices International Conference (MAPLD'04)*, 2004.
25. Atmel. *5K - 50K Gates Coprocessor FPGA with FreeRAM*, April 2002.
26. Ievgenii Bakhmach, Alexander Siora, Victor Tokarev, Sergey Reshetitsky, and Volodymyr Bezsalyi. Implementation Principles of FPGA-based ESFAS for Kozloduy NPP. In *Proceedings of the 6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC&HMIT 2009)*. American Nuclear Society, April 2009.
27. R.C. Baumann. Radiation-induced Soft Errors in Advanced Semiconductor Technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305 – 316, September 2005.
28. M. Bellato, M. Ceschia, M. Menichelli, A. Papi, J. Wyss, and A. Paccagnella. Ion beam testing of sram-based fpga's. In *Proceedings of the 6th European Conference on Radiation and Its Effects on Components and Systems*, pages 474 – 480, sep. 2001.
29. Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An Overview of SAL. In *Proceedings of the Fifth NASA Langley Formal Methods Workshop (LFM 2000)*, pages 187–196, 2000.
30. C. Bernardeschi, L. Cassano, M.G.C.A. Cimino, and A. Domenici. Application of a genetic algorithm for testing SEUs in SRAM-FPGA Systems. In *Proceedings of the 6th HiPEAC Workshop on Reconfigurable Computing (WRC2012)*, 2012.
31. C. Bernardeschi, L. Cassano, and A. Domenici. Failure Probability and Fault Observability of SRAM-FPGA Systems. In *International Conference on Field Programmable Logic and Applications (FPL2011)*, pages 385 –388, sept. 2011.
32. C. Bernardeschi, L. Cassano, and A. Domenici. Failure Probability of SRAM-FPGA Systems with Stochastic Activity Networks. In *Proceedings of the 14th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, April 2011.
33. C. Bernardeschi, L. Cassano, and A. Domenici. SEU-X: a SEu Un-uXecitbility prover for SRAM-FPGAs. In *Proceedings of the 18th IEEE International On-Line Testing Symposium (IOLTS2012)*, June 2012.
34. C. Bernardeschi, L. Cassano, A. Domenici, G. Gennaro, and M. Pasquariello. Simulated Injection of Radiation-Induced Logic Faults in FPGAs. In *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011.
35. C. Bernardeschi, L. Cassano, A. Domenici, and L. Sterpone. Accurate Simulation of SEUs in the Configuration Memory of SRAM-based FPGAs. In *Proceedings of the 25th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, Octobr 2012.
36. C. Bernardeschi, L. Cassano, A. Domenici, and L. Sterpone. Unexcitability Analysis of SEUs Affecting the Routing Structure of SRAM-based FPGAs. In *Accepted at the 23rd Great Lakes Symposium on Very Large Scale of Integration (GLSVLSI2013)*, May 2013.
37. Cinzia Bernardeschi, Luca Cassano, Andrea Domenici, and Luca Sterpone. Accurate Simulation of SEUs in the Configuration Memory of SRAM-based FPGAs. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT 2012)*, 2012.

38. BlueTeam. Autonomous motorcycle platform and navigation, 2005. Darpa Grand Challange.
39. Miljko Bobrek, Richard T. Wood, Donald Bouldin, and Michael E. Waterman. FPGA Design Practices for I&C in Nuclear Power Plants. In *Proceedings of the 6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC&HMIT 2009)*. American Nuclear Society, April 2009.
40. J. Borecky, P. Kubalik, and H. Kubatova. Reliable Railway Station System Based on Regular Structure Implemented in FPGA. In *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD '09)*, pages 348 –354, 2009.
41. A. Bosio and G. Di Natale. LIFTING: A Flexible Open-Source Fault Simulator. In *Proceedings of the 17th Asian Test Symposium (ATS '08)*, pages 35 –40, nov. 2008.
42. A. Brown and P. Olson. Urban/indoor navigation using network assisted gps. In *Proceedings of ION 61st Annual Meeting*, June 2005.
43. W. Calienes Bartra and R. Reis. Set and seu simulation toolkit for labview. In *12th European Conference on Radiation and Its Effects on Components and Systems (RADECS2011)*, pages 829 –836, sept. 2011.
44. Carl Carmichael, Earl Fuller, Joe Fabula, and Fernanda D. Lima. Proton testing of SEU mitigation methods for the Virtex FPGA. In *Proceedings of the IEEE Microelectronics Reliability and Qualification Workshop*, Pasadena, CA, December 2001.
45. James A. Cercone, Michael A. Beims, and Kenneth G. McGill. Verification and Validation of Programmable Logic Devices. In *Proceedings of the 7th Military and Aerospace Programmable Logic Devices International Conference (MAPLD'04)*. National Aeronautics and Space Administration (NASA), September 2004.
46. M. Ceshia, M. Bellato, A. Paccagnella, S. C. Lee, C. Wan, A. Kaminski, M. Menichelli, A. Papi, and J. Wyss. Ion beam testing of Altera Apex FPGAs. In *Proceedings of the 2002 IEEE Radiation Effects Data Workshop*, pages 45–50, July 2002.
47. Gang Chen and Li Guo. The fpga implementation of kalman filter. In *ISCGAV'05: Proceedings of the 5th WSEAS International Conference on Signal Processing, Computational Geometry & Artificial Vision*, pages 61–65, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).
48. Henrik B. Christophersen, Wayne J. Pickell, Adrian A. Koller, Suresh K. Kannan, and Eric N. Johnson. Small adaptive flight control systems for uavs using fpga/dsp technology. American Institute of Aeronautics and Astronautic.
49. G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius modeling tool. In *9th Int. Workshop on Petri Nets and Performance Models*, pages 241–250, Aachen, Germany, September 2001. IEEE Computer Society Press.
50. Srdjan Coric, Miriam Leeser, Eric Miller, and Marc Trepanier. Parallel-beam backprojection: an FPGA implementation optimized for medical imaging. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays (FPGA '02)*, pages 217–226, New York, NY, USA, 2002. ACM.
51. F. Corno, F. Cumani, and G. Squillero. Exploiting auto-adaptive μgp for highly effective test programs generation. In *Proceedings of the 5th international conference on Evolvable systems: from biology to hardware*, ICES'03, pages 262–273, Berlin, Heidelberg, 2003. Springer-Verlag.
52. F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. Automatic test program generation for pipelined processors. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 736–740, New York, NY, USA, 2003. ACM.
53. F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. Fully automatic test program generation for microprocessor cores. In *Proceedings of the conference on Design, Au-*

tomation and Test in Europe - Volume 1, DATE '03, pages 11006–, Washington, DC, USA, 2003. IEEE Computer Society.

54. F. Corno, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. GATTO: a genetic algorithm for automatic test pattern generation for large synchronous sequential circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):991 – 1000, aug 1996.
55. F. Corno, E. Sanchez, M. Sonza Reorda, and G. Squillero. Code generation for functional validation of pipelined microprocessors. *J. Electron. Test.*, 20(3):269–278, June 2004.
56. F. Corno, M. Sonza Reorda, and G. Squillero. RT-Level ITC'99 Benchmarks and First ATPG Results. *IEEE Des. Test*, 17:44–53, July 2000.
57. Fulvio Corno, Ernesto Sánchez, Matteo Sonza Reorda, and Giovanni Squillero. Automatic test program generation: A case study. *IEEE Des. Test*, 21(2):102–109, March 2004.
58. Altera Corporation. Altera automotive products.
59. Lattice Semiconductor Corporation. Lattice automotive - accelerated time-to-market with low cost programmable logic.
60. Lattice Semiconductor Corporation. Ready-to-use ethernet portfolio.
61. Lattice Semiconductor Corporation. Implementing wimax ofdm timing and frequency offset estimation in lattice fpgas, November 2005.
62. J. Cromwell, G. Paparisto, and K. M. Chugg. On the design and hardware demonstration of a robust, high-speed frequency-hopped radio for severe battlefield channels. In *Proceedings of the 2002 IEEE Military Communications Conference,*, October 2002.
63. Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The Möbius framework and its implementation. *IEEE Trans. Softw. Eng.*, 28(10):956–969, 2002.
64. J. Deepakumara, H.M. Heys, and R. Venkatesan. Fpga implementation of md5 hash algorithm. In *Electrical and Computer Engineering, 2001. Canadian Conference on*, volume 2, pages 919–924 vol.2, 2001.
65. R. Dobias and H. Kubatova. Fpga based design of the railway's interlocking equipments. In *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, pages 467–473, Aug.-3 Sept. 2004.
66. P.E. Dodd and L.W. Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on Nuclear Science*, 50(3):583 – 602, june 2003.
67. eds. S. Tavares and H. Meijer, editor. *Fast DES Implementations for FPGAs and its Application to a Universal Key-Search Machine*. in 5th Annual Workshop on Selected Areas in Cryptography (SAC '98) vol. LNCS 1556, Springer-Verlag, August 1998.
68. A.J. Elbirt and C. Paar. An fpga implementation and performance evaluation of the serpent block cipher. In *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 33–40, New York, NY, USA, 2000. ACM.
69. European Committee for Electrotechnical Standardization (CENELEC). EN 50128: Railway applications - Communications, signaling and processing systems - Software for railway control and protection systems, April 2002.
70. European Committee for Electrotechnical Standardization (CENELEC). EN 50129: Railway applications - Communications, signaling and processing systems - Safety related electronic systems for signaling, February 2003.
71. European Space Agency (ESA). *Space environments and effects*, April 2007.
72. European Space Agency (ESA). *Single Event Effects (SEE) Mechanism and Effects*, June 2009.
73. A. Fernàndez-Leòn, A. Pouponnot, and S. Habinc. ESA FPGA Task Force: Lessons Learned. In *Proceedings of the 5th Military and Aerospace Programmable Logic Devices International Conference (MAPLD'02)*. National Aeronautics and Space Administration (NASA), September 2002.

74. F. Ferrandi, A. Fin, F. Fummi, and D. Sciuto. Functional test generation: Overview and proposal of a hybric genetic approach. In Rolf Drechsler and Nicole Drechsler, editors, *Evolutionary Algorithms in Circuit Design*, pages 105–142. Kluwer, 2003.
75. Radio Technical Commission for Aeronautics (RTCA). DO-178B Software Considerations in Airborne Systems and Equipment Certification, December 1992.
76. Radio Technical Commission for Aeronautics (RTCA). DO-254 Design Assurance Guidance for Airborne Elecronic Hardware, April 2000.
77. European Cooperation for Space Standardization (ECSS). Q-ST-60-02C Space product assurance: ASIC and FPGA development, July 2008.
78. H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. Comput.*, 32(12):1137–1144, December 1983.
79. Earl Fuller, Michael Caffrey, Phil Blain, Carl Carmicheal, Noor Khalsa, and Anthony Salazar. Radiation test results of the Virtex FPGA and ZBT SRAM for space based reconfigurable computing. In *Proceedings of the 2nd Military and Aerospace Programmable Logic Devices International Conference (MAPLD'99)*, 1999.
80. Earl Fuller, Michael Caffrey, Anthony Salazar, Carl Carmichael, and Joe Fabula. Radiation testing update, seu mitigation, and availability analysis of the virtex fpga for space re-configurable computing", presented at the ieee nuclear and space radiation effects conference. In *Proceedings of the 3rd Military and Aerospace Programmable Logic Devices International Conference (MAPLD'00)*, 2000.
81. M. Gen and R. Cheng. *Genetic Algorithms and Engineering Design*. John Wiley & Sons., 1997.
82. W. Gibbons and H. Ames. Use of FPGAs in Critical Space Flight Applications - A Hard Lesson. In *Proceedings of the Military and Aerospace Applications of the Programmable Devices and Technologies Conference*. National Aeronautics and Space Administration (NASA), 1999.
83. Iain Goddard and Marc Trepanier. The role of FPGA–based processing in medical imaging. Technical report, VMEbus Systems, April 2003.
84. P. Goel. RAPS Test Pattern Generator. *IBM Technical Disclosure Bulletin*, 21(7):2787–2791, 1978.
85. P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Trans. Comput.*, 30(3):215–222, March 1981.
86. G. R. Goslin. A guide to using field programmable gate arrays for application-specific digital signal processing performance. In *Proceedings of SPIE*, volume vol. 2914, pages p321–331, 1995.
87. F. Gosset, F.-X. Standaert, and J.-J. Quisquater. Fpga implementation of squash. In *In Pro-ceedings of the 29th Symposium on Information Theory in the Benelux*, 2008.
88. Government Microcircuit Applications and Critical Technology Conference. *Accelerating defense applications using high performance reconfigurable computing*, April 2003.
89. P. Graham, M. Caffrey, J. Zimmerman, D. E. Johnson, P. Sundararajan, and C. Patterson. Consequences and Categories of SRAM FPGA Configuration SEUs. In *Proceedings of the 6th Military and Aerospace Applications of Programmable Logic Devices (MAPLD'03)*, September 2003.
90. Paul Graham and Brent Nelson. Fpga-based sonar processing. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 201–208, New York, NY, USA, 1998. ACM.
91. Tim Grembowski, Roar Lien, Kris Gaj, Nghi Nguyen, Peter Bellows, Jaroslav Flidr, Tom Lehman, and Brian Schott. Comparative analysis of the hardware implementations of hash functions sha-1 and sha-512. In *ISC '02: Proceedings of the 5th International Conference on Information Security*, pages 75–89, London, UK, 2002. Springer-Verlag.
92. Michael Gschwind, Valentina Salapura, and Dietmar Maurer. Fpga prototyping of a risc processor core for embedded applications. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(2):241–250, 2001.

93. Rohit Gulati and Joanne Bechta Dugan. A modular approach for analyzing static and dynamic fault trees. In *Annual Reliability and Maintainability Symposium*, pages 57–63. IEEE Computer Society Press, 1997.
94. Daniel González Gutiérrez. Single event upsets simulation tool functional description. Technical Report TEC-EDM/DGG-SST2, ESA-ESTEC, 2000.
95. Sandi Habinc. Lessons Learned from FPGA Developments, September 2002. Technical Report. Gaisler Research.
96. Grant Hampson. A possible 100 msps altera fpga fft processor, March 2002.
97. Georg Hanak and Ralph Mende. Using FPGAs In Automotive Radar Sensors. Altera Corporation.
98. Scott Hauck. The roles of fpgas in repreogrammable systems. *IEEE*, 86(4):615–639, April 1998.
99. J. Henaut, D. Dragomirescu, and R. Plana. FPGA Based High Date Rate Radio Interfaces for Aerospace Wireless Sensor Systems. In *Proceedings of the Fourth International Conference on Systems (ICONS '09)*, pages 173 –178, 2009.
100. O. Heron, T. Arnaout, and H.-J. Wunderlich. On the reliability evaluation of sram-based fpga designs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'05)*, pages 403 – 408, August 2005.
101. J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975.
102. W.K. Huang, F.J. Meyer, N. Park, and F. Lombardi. Testing Memory Modules in SRAM-based Configurable FPGAs. In *Proceedings of the International Workshop on Memory Technology, Design and Testing*, pages 79 –86, aug 1997.
103. Mike Hutton and Roger May. Programmable Solutions for Automotive Systems. Altera Corporation.
104. International Atomic Energy Agency (IAEA). NS-G-1.1: Software for Computer Based Systems Important to Safety in Nuclear Power Plants, 2000. IAEA Safety Standards Series.
105. International Atomic Energy Agency (IAEA). NS-G-1.3: Instrumentation and Control Systems Important to Safety in Nuclear Power Plants, 2002. IAEA Safety Standards Series.
106. International Organization for Standardization (ISO). 26262-5: Road vehicles - Functional safety - Part 5. Product development: hardware level, December 2009. Draft.
107. International Organization for Standardization (ISO). 26262-6: Road vehicles - Functional safety - Part 6. Product development: software level, December 2009. Draft.
108. Charalambos Ioannides and Kerstin I. Eder. Coverage-directed test generation automated by machine learning – a review. *ACM Trans. Des. Autom. Electron. Syst.*, 17(1):7:1–7:21, January 2012.
109. Massimiliano Leone Itria. Progetto e realizzazione di un traduttore per il linguaggio EDIF orientato a sistemi FPGA. Master's thesis, Department of Information Engineering, University of Pisa, Italy, 2011.
110. J. Hagemeyer, A. Hilgenstein, D. Jungewelter, D. Cozzi, C. Felicetti, U. Rückert, S Korf, M. Köster, F. Margaglia, M. Porrmann, F. Dittmann, M. Ditze, J. Harris, L. Sterpone, J. Ilstad. A Scalable Platform for Run-time Reconfigurable Satellite Payload Processing. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2012)*, 2012.
111. Rau Jiann-Chyi, Ho Ying-Fu, and Wu Po-Han. A novel reseeding mechanism for pseudo-random testing of vlsi circuits. In *IEEE International Symposium on Circuits and Systems (ISCAS 2005)*, pages 2979 – 2982 Vol. 3, may 2005.
112. R. Katz, K. LaBel, J.J. Wang, B. Cronquist, R. Koga, S. Penzin, and G. Swift. Radiation effects on current field programmable technologies. *IEEE Transactions on Nuclear Science*, 44(6):1945 –1956, dec 1997.
113. Kevin M. Kitagawa. At the heart of consumer and automotive innovation.

114. Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, 26(2), February 2007.
115. Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Foundations and Trends Electronic Design Automation*, 2(2):135–253, February 2008.
116. M. Lanuzza, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello. Exploiting Self-Reconfiguration Capability to Improve SRAM-based FPGA Robustness in Space and Avionics Applications. *ACM Transactions on Reconfigurable Technology and Systems*, 4:8:1–8:22, December 2010.
117. Lattice Semiconductor Corporation. *LatticeXP Family Handbook*, November 2007.
118. Lattice Semiconductor Corporation. *LatticeECP2/M Family Handbook*, March 2009.
119. Lattice Semiconductor Corporation. *LatticeECP3 Family Handbook*, November 2009.
120. A. Ledeczi, P. Volgyesi, M. Maroti, G. Simon, G. Balogh, A. Nadas, B. Kusy, S. Dora, and G. Pap. Multiple simultaneous acoustic source localization in urban terrain. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 491–496, April 2005.
121. Guy G. F. Lemieux, Stephen D. Brown, and Daniel Vranesic. On two-step routing for FPGAS. In *Proceedings of the 1997 international symposium on Physical design (ISPD '97)*, pages 60–66, New York, NY, USA, 1997. ACM.
122. Jianchun Li, Christos Papachristou, and Raj Shekhar. An FPGA-based computing platform for real-time 3D medical imaging and its application to cone-beam CT reconstruction. *Journal of Imaging Science and Technology*, 49:237–245, 2005.
123. Sheac Yee Lim and Andrew Crosland. Implementing fft in an fpga co-processor. In *In The International Embedded Solutions Event (GSPx)*, pages 27–30, 2004.
124. Jan-Ruei Lin and Shih-Ching Ou. Design of encryption chips using the blowfish algorithm. Master's thesis, Department of Electrical engineering National Central University Chungli, Taiwan, 1999.
125. D.E. Long, M.A. Iyer, and M. Abramovici. FILL and FUNI: Algorithms to Identify Illegal States and Sequential Untestable Faults. *ACM Transaction on Design Automation of Electronic Systems*, 5(3):632–657, 2000.
126. Jing Lu and John Lockwood. Ipsec implementation on xilinx virtex-ii pro fpga and its application. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 158.2, Washington, DC, USA, 2005. IEEE Computer Society.
127. D.M. MacQueen, D.M. Gingrich, N.J. Buchanan, and P.W. Green. Total ionizing dose effects in a sram-based fpga. In *Radiation Effects Data Workshop*, pages 24 –29, 1999.
128. A. Mazzeo, L. Romano, G. P. Saggese, and N. Mazzocca. Fpga-based implementation of a serial rsa processor. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10582, Washington, DC, USA, 2003. IEEE Computer Society.
129. Tapan A. Mehta. How FPGAs Enable Automotive Systems, 2005. Altera Corporation.
130. Mentor Graphics Corporation. *ModelSim SE Reference Manual*, 2008.
131. Z. Michalewicz. *Genetic Algorithms Plus Data Structures Equals Evolution Programs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1994.
132. Microsemi. *eX Automotive Family FPGAs Datasheet*, June 2006.
133. Microsemi. *MX Automotive Family FPGAs Datasheet*, May 2006.
134. Microsemi. *SX-A Family FPGAs Datasheet*, February 2007.
135. Microsemi. *IGLOO Low-Power Flash FPGAs Handbook*, November 2009.
136. Microsemi. *Radiation-Tolerant ProASIC3 Handbook*, November 2009.
137. Microsemi. *RProASIC3 Handbook*, October 2009.
138. Microsemi. *RProASIC3E Handbook*, August 2009.
139. Microsemi. *RTAX-DSP Radiation-Tolerant FPGAs, Advance Product Brief*, November 2009.

140. Microsemi. *RTAX-S/SL RadTolerant FPGAs, Detailed Specifications*, May 2009.
141. Microsemi. *ProASIC Flash Family FPGAs Datasheet*, January 2010.
142. Alexander Miczo. *Digital Logic Testing and Simulation*. John Wiley & Sons., 2003.
143. E. Monmasson and M.N. Cirstea. Fpga design methodology for industrial control systems—a review. *Industrial Electronics, IEEE Transactions on*, 54(4):1824–1842, Aug. 2007.
144. National Aeronautics and Space Administration (NASA). *Natural Space Radiation Effects on Technology*, November 2000.
145. National Aeronautics and Space Administration (NASA). *The Natural Space Radiation Hazard*, November 2000.
146. National Aeronautics and Space Administration (NASA). *The Sun-Earth Connection: Heliophysics Solar Storm and Space Weather*, April 2012.
147. M.J. O'Dare and T. Arslan. Hierarchical test pattern generation using a genetic algorithm with a dynamic global reference table. In *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*, pages 517–523, sep 1995.
148. T.R. Oldham and F.B. McLean. Total ionizing dose effects in mos oxides and devices. *IEEE Transactions on Nuclear Science*, 50(3):483 – 499, june 2003.
149. Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Hochschule Darmstadt, 1961.
150. Proc. Third Advanced Encryption Standard Candidate Conf. *An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists*, 2000.
151. J. Raik, H. Fujiwara, R. Ubar, and A. Krivenko. Untestable Fault Identification in Sequential Circuits Using Model-Checking. In *Proceedings of the 17th Asian Test Symposium (ATS'08)*, pages 21–26, 2008.
152. J. Raik, A. Rannaste, M. Jenihhin, T. Viilukas, R. Ubar, and H.; Fujiwara. Constraint-Based Hierarchical Untestability Identification for Synchronous Sequential Circuits. In *Proceedings of the 16th European Test Symposium (ETS'11)*, pages 147–152, 2011.
153. J. Raik, R. Ubar, A. Krivenko, and M. Kruus. Hierarchical Identification of Untestable Faults in Sequential Circuits. In *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD'07)*, 2007.
154. Joydeep Ray and James C. Hoe. High-level modeling and fpga prototyping of microprocessors. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 100–107, New York, NY, USA, 2003. ACM.
155. M. Rebaudengo, M. Sonza Reorda, and M. Violante. A new functional fault model for FPGA application-oriented testing. In *Proceedings of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002)*, pages 372 – 380, 2002.
156. M. Renovell, J.M. Portal, P. Faure, J. Figueras, and Y. Zorian. Analyzing the Test Generation Problem for an Application-Oriented Test of FPGAs. In *Proceedings of the IEEE European Test Workshop*, pages 75 –80, 2000.
157. M. Renovell, J.M. Portal, J. Figuras, and Y. Zorian. Minimizing the Number of Test Configurations for Different FPGA Families. In *Proceedings of the Eighth Asian Test Symposium (ATS '99)*, pages 363 –368, 1999.
158. M. Riaz and H.M. Heys. The fpga implementation of the rc6 and cast-256 encryption algorithms. In *Electrical and Computer Engineering, 1999 IEEE Canadian Conference on*, volume 1, pages 367–372 vol.1, 1999.
159. J. Paul Roth. Diagnosis of Automata Failures: A Calculus and a Method. *IBM Journal of Research and Development*, 10(4):278 –291, July 1966.

160. J. Paul Roth, Willard G. Bouricius, and Peter R. Schneider. Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits. *Electronic Computers, IEEE Transactions on*, EC-16(5):567 –580, oct. 1967.
161. C. Rousselle, M. Pflanz, A. Behling, T. Mohaupt, and H.T. Vierhaus. A register-transfer-level fault simulator for permanent and transient faults in embedded processors. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, page 811, 2001.
162. K.Y. Rozier. Linear Temporal Logic Symbolic Model Checking. *Computer Science Review*, 5(2):163 – 203, 2011.
163. M. Rozkovec, J. Jenicek, and O. Novak. Application Dependent FPGA Testing Method. In *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD10)*, pages 525 –530, sept. 2010.
164. Elizabeth Rudnick, John G. Holm, Daniel G. Saab, and Janak H. Patel. Application of simple genetic algorithms to sequential circuit test generation. In *Proc. European Design and Test Conf*, pages 40–45, 1994.
165. E.M. Rudnick, J.H. Patel, G.S. Greenstein, and T.M. Niermann. A Genetic Algorithm Framework for Test Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(9):1034 –1044, sep 1997.
166. D. Runje and M. Kovac. Universal strong encryption fpga core implementation. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 923–924, Washington, DC, USA, 1998. IEEE Computer Society.
167. G. De Ruvo, P. De Ruvo, F. Marino, G. Mastronardi, P.L. Mazzeo, and E. Stella. A FPGA-Based Architecture for Automatic Hexagonal Bolts Detection in Railway Maintenance. In *Proceedings of the Seventh International Workshop on Computer Architecture for Machine Perception (CAMP '05)*, pages 219–224, Washington, DC, USA, 2005. IEEE Computer Society.
168. D. G. Saab, Y. G. Saab, and J. A. Abraham. Cris: a test cultivation program for sequential vlsi circuits. In *1992 IEEE/ACM international conference proceedings on Computer-aided design*, ICCAD '92, pages 216–219, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
169. A. Samarah, A. Habibi, S. Tahar, and N. Kharma. Automated coverage directed test generation using a cell-based genetic algorithm. In *High-Level Design Validation and Test Workshop, 2006. Eleventh Annual IEEE International*, pages 19 –26, nov. 2006.
170. W. Sanders and J. Meyer. Stochastic activity networks: formal definitions and concepts. In E. Brinksma, H. Hermanns, and J.P. Katoen, editors, *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *Lecture Notes in Computer Science*, pages 315–343. Springer Berlin / Heidelberg, 2001.
171. William H. Sanders and John F. Meyer. A unified approach for specifying measures of performance, dependability, and performability. In Algirdas Avižienis, Herman Kopetz, and Jean-Claude Laprie, editors, *Dependable Computing for Critical Applications*, pages 215–237. Springer-Verlag Heidelberg, 1991.
172. S. Schulz, G. Beltrame, and D. Merodio-Codinachs. Smart behavioral netlist simulation for seu protection verification. In *9th European Conference on Radiation and Its Effects on Components and Systems (RADECS2008)*, pages 406 –411, sept. 2008.
173. J. She and J. Jiang. Application of FPGA to Shutdown System No.1 in Candu. In *Proceedings of the 6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC&HMIT 2009)*. American Nuclear Society, 2009.
174. Yu.A. Skobtsov and V.Yu. Skobtsov. Evolutionary approach to test generation of sequential digital circuits with multiple observation time strategy. In *Proceedings of the East-West Design Test Symposium (EWDTS10)*, pages 286 –291, sept. 2010.

175. Andreas Söderberg, Jacques Hérard, and Lars Bo Mortensen. Guideline for Design and Safety Validation of Safety-Critical Functions Realized with Hardware Description Language, May 2005. Technical Report. NORDTEST: a Nordic Innovation Center Brand.
176. L. Sterpone and M. Violante. A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs. *IEEE Transactions on Nuclear Science*, 52(6):2217–2223, December 2005.
177. L. Sterpone and M. Violante. Analysis of the robustness of the TMR architecture in SRAM-based FPGAs. *IEEE Transactions on Nuclear Science*, 52(5):1545–1549, oct. 2005.
178. M. Straka, J. Kastil, and Z. Kotasek. SEU Simulation Framework for Xilinx FPGA: First Step towards Testing Fault Tolerant Systems. In *Proceedings of the 14th Euromicro Conference on Digital System Design (DSD2011)*, pages 223–230, 31 2011-sept. 2 2011.
179. C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici. Built-in Self-Test of FPGA Interconnect. In *Proceedings of the International Test Conference*, pages 404 –411, oct 1998.
180. M. Tahoori. Application-Dependent Testing of FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(9):1024 –1033, 2006.
181. Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 28:7–27, 2000.
182. D. Tille and R. Drechsler. A Fast Untestability Proof for SAT-based ATPG. In *Proceedings of the 12th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS'09)*, pages 38–43, 2009.
183. Keith Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays (FPGA '04)*, pages 171–180, New York, NY, USA, 2004. ACM.
184. Keith D. Underwood and K. Scott Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04)*, pages 219–228, Washington, DC, USA, 2004. IEEE Computer Society.
185. Moshe Y. Vardi. An Automata-Theoretic Approach to Linear Temporal Logic. *Logics for Concurrency: Structure versus Automata*, 1043:238–266, 1996.
186. M. Violante, N. Battezzati, and L. Sterpone. *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Springer Science & Business Media, 2011.
187. Kenneth D. Wagner, Cary K. Chin, and Edward J. McCluskey. Pseudorandom testing. *IEEE Trans. Comput.*, 36(3):332–343, March 1987.
188. J.J. Wang. Radiation effects in FPGAs. In *Proceedings of the 9th Workshop on Electronics for LHC Experiments*, pages 34–43, October 2003.
189. S. Wilton. Architecture and Algorithms for Field-Programmable Gate Arrays with Embedded Memory. Master's thesis, Department of Electrical and Computer Engineering University of Toronto, Canada, 1997.
190. V. Winkler, J. Detlefsen, U. Siart, J. Buchler, and M. Wagner. FPGA-based Signal Processing of an Automotive Radar Sensor. In *Proceedings of the First European Radar Conference (EURAD)*, pages 245 –248, 2004.
191. M. Wirthlin, E. Johnson, N. Rollins, M. Caffrey, and P. Graham. The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, pages 133–142, april 2003.
192. Roland E. Wunderlich and James C. Hoe. In-system fpga prototyping of an itanium microarchitecture. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 255–255, New York, NY, USA, 2004. ACM.
193. Xilinx. Wireless solutions.
194. Xilinx. Xilinx automotive – flexible solutions beyond silicon.

195. Xilinx. *XC2064/XC2018 Logic Cell Array*, 1985.
196. Xilinx. *Broadcast Audio/Video Connectivity with Virtex-5 FPGAs*, 2009.
197. Xilinx. *Spartan-6 Family Overview*, November 2009.
198. Xilinx. *Virtex-5 Family Overview*, November 2009.
199. Xilinx. *Virtex-6 Family Overview*, January 2010.
200. Xilinx. *The Xilinx Virtex-7 FPGA Family: unleashing performance abd innovation with high-density, low-power 28nm technology*, 2012.
201. Haissam Ziade, Rafic A. Ayoubi, and Raoul Velazco. A Survey on Fault Injection Techniques. *The International Arab Journal of Information Technolology*, pages 171–186, 2004.