Akademia Górniczo-Hutnicza
im. Stanisława Staszica
Wydział Informatyki, Elektroniki i Telekomunikacji

Rozprawa doktorska

TESTING OF SOFT PROCESSOR CORES IMPLEMENTED IN FPGA

Mgr inż. Mariusz Węgrzyn

Promotor (supervisor): dr hab. inż Ernest Jamro, prof. AGH

Promotor pomocniczy (auxuliary supervisor):

dr. inż. Agnieszka Dąbrowska-Boruch

Kraków 2020

Serdeczne podziękowania dla Doktora habilitowanego Ernesta Jamro,
Profesora AGH
za nieocenioną pomoc, porady podczas badań oraz kierowanie tą pracą.

Thank you to my supervisor D.Sc. Ernest Jamro,
Professor of AGH University of Science and Technology
for invaluable help; advices at improvement of researches,
guidance and feedback throughout this dissertation.

„Sukces wydaje się być w dużej mierze kwestią wytrwania”

OUTLINE

List of acronyms

ADL	Architecture Description Language
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generator
BIST	Built-in Self-Test
BJT	Bipolar Junction Transistor
BRAM	Block Random-Access Memory
CEU	Code Emulated Upset
CIP	Craig Interpolation Prover
CLB	Configurable Logic Block
CMC	Configurable Memory Cell
CMOS	Complementary Metal-Oxide Semiconductor
CRC	Cyclic Redundancy Check
DAG	Direct Acyclic Graph
DFF	D-type flip flop
DSP	Digital Signal Processing
DUT	Device Under Test
FC	Fault Coverage
FCmax	Maximal Fault Coverage
FF	Flip-flop
FI	Fault Injection
FIHU	Fault Injection Hardware Unit
FinFET	Fin field-effect transistor
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FTI	Fixed Test Instruction
GPIO	General-purpose input/output
HDL	Hardware Description Language

IC	Integrated Circuit
IOB	Input Output Block
IP	Intellectual Property
IRST	Instruction Randomization Self-Test
ISA	Instruction Set Architecture
JFET	Junction Field Effect Transistor
JTAG	Joint Test Action Group
LET	Linear Energy Transfer
LFSR	Linear Feedback Shift Register
LUT	Look-up Table
MIHST	Microprocessor Hardware Self-Test
MIPS	Microprocessor without Interlocked Pipelined Stages
MIS	Modifiable Instruction Storage
MOSFET	Metal-Oxide Semiconductor Field-Effect Transistor
NASA	National Aeronautics and Space Administration
PC	Program Counter
PIP	Programmable Interconnect Point
RAM	Random-Access Memory
RTL	Register Transfer Level
SBST	Software Based Self Test
SEB	Single Event Burn out
SEE	Single Event Effect
SEGR	Single Event Gate Rupture
SEL	Single Event Latch up
SET	Single Event Transient
SEU	Single Event Upset
SoC	System on Chip
SoPC	System on Programmable Chip
SOS	System on Sapphire
SRAM	Static Random-Access Memory
SSB	Single Switch Box
STF	Single Transient Fault

TMR	Triple Modular Redundancy
TRCD	Test Response Compression Device
TS	Test Sequence
Qcrit	Critical charge
VHDL	Very High Description Language

Abstract

Soft processor cores, which are widely used in SRAM-based FPGA (Field Programmable Gate Arrays) applications, are candidates for SEU-induced (Single Event Upset) faults and therefore these cores need to be thoroughly tested. In user applications, processor cores are normally tested by executing some kind of functional test in which the individual processor's instructions are tested with a set of deterministic test patterns, and the results are then compared with the stored reference values. For practical limitations the number of test patterns and corresponding results are usually small, which inherently leads to low fault coverage.

The proposed approach is based on a data-sensitive path with slightly different meaning as known from bibliography. In this work the author has developed a concept that combines the whole instruction-set test into a bijective test sequence. According to this strict rule a novel test-sequence generation principle was introduced, where the test sequence requires one-to-one bijective correspondence between the input test pattern and the result. In this way the author has activated high percent of data sensitive paths. Hence, the program composed from bijective blocks achieved significantly better fault coverage (85,6%) than well-known computing application or test programs with simpler architectures. Definitively, the author has achieved the best fault coverage (94,76 %) by creating bijective test program, which generates simultaneously complete cycle of local test vectors.

The approach is illustrated by an experimental case study and evaluated by simulating faults in the HDL (hardware description language) description of the processor core. In order to determine the fault coverage of SEU-induced faults a model of fault injection must be provided. As an alternative to the statistical-based radiation tests, an original simulation-based solution was invented by the author. The faults in an HDL description of a system are modeled by automated modifying the individual bits in LUT (Look Up Table) memory. Behavior of each functional block is described by an HDL model, after a fault has been injected. Their HDL descriptions reflect the FPGA structure in order to efficiently use the FPGA resources. One of the most important novelty introduced hereby is a novel model of injected faults. Benefits of novelty proposed in this work are double, because in this way it is possible to model natural SEU faults in LUTs, which lead to different implementations of logical functions as these intended. The second benefit is, that these faults can be interpreted in particular cases as a stuck-at „0" or stuck-at „1" faults at inputs or output of LUTs, so the FPGA routing resources are also simulated. The injected faults are equivalent to SEUs. The HDL model reflects the change of configuration, which is a consequence of the SEU effect. Using this model the author has elaborated complete system to evaluation test programs dedicated to test of processor cores implemented in FPGAs.

The further part of the dissertation is devoted to development of the test optimization methods. This part proves, that the set of test vectors can be minimized nine times. Three optimization strategies are presented herein. These methods saved up memory resources and shortened testing time. The author has proposed an approach to testing of individual blocks of processor and optimization of sets of local test vectors.

The last chapter describes the problem of logical and hardware redundancies, which make impossible achievement of 100% fault coverage for any complex system implemented in FPGA. This chapter presents methods, examples and results of detailed author's researches in this matter. A comparison of these optimization methods is made at the end of the dissertation.

Streszczenie

Rdzenie soft procesorowe, które są szeroko stosowane w układach programowalnych FPGA (Field Programmable Gate Arrays) bazujących na pamięciach SRAM, są szczególnie podatne na błędy indukowane typu SEU (Single Event Upset) i dlatego soft procesory powinny być gruntownie testowane. W aplikacjach użytkowych, rdzeń procesora jest normalnie testowany poprzez wykonywanie testu funkcjonalnego, podczas którego poszczególne instrukcje procesora są weryfikowane zestawem deterministycznych wzorców testowych i rezultaty są porównywane z zapisanymi referencyjnymi wartościami. Z powodu praktycznych ograniczeń, liczba wzorców testowych i odpowiadających im rezultatów jest zwykle mała, co naturalnie prowadzi do niskiego pokrycia błędów.

Nowe podejście proponowane przez autora bazuje na ścieżce wrażliwej na dane, która ma nieco inną interpretacje jak ta, znana z bibliografii. W tej pracy autor rozwinął koncept który formuje bijektywną sekwencję testową złożoną z prawie wszystkich instrukcji procesora. Efektywność tego rozwiązania jest osiągnięta poprzez twardą zasadę, według której nowatorski sposób generacji sekwencji testowej wymaga bijektywnej relacji ,,jeden do jednego" pomiędzy wejściowymi wzorcami testowymi oraz rezultatami. W ten sposób autor aktywował duży procent ścieżek wrażliwych na dane. Związane z tym lepsze pokrycie błędów (85,6%) zostało osiągnięte przez program złożony z bijektywnych bloków (85,6%). Zdecydowanie najlepsze pokrycie błędów (94,76%) autor osiągnął kreując bijektywny program testowy, który generuje jednocześnie pełny cykl lokalnych wektorów.

Podejście autora ilustruje eksperymentalny przypadek studyjny oraz ewaluację poprzez symulację błędów w rdzeniu procesora opisanym w języku HDL (Hardware Description Language). W celu określenia pokrycia błędów wymagane jest określenie sposobu wstrzykiwania błędów. Jako alternatywa do testów statystycznych bazujących na eksponowaniu FPGA na promieniowanie jonizujące, zostało opracowane oryginalne rozwiązanie oparte na symulacjach. Błędy w opisie HDL są modelowane poprzez zautomatyzowaną modyfikację indywidualnych bitów pamięci LUT (Look Up Table). Zaproponowana zautomatyzowana symulacja wszystkich możliwych 1804 wstrzykniętych błędów jest bardzo bliska rzeczywistych błędów typu SEU co stanowi znaczące osiągnięcie tej pracy. Nowatorski model błędów przynosi podwójne korzyści: po pierwsze w ten sposób modelowane są błędy indukowane w pamięciach LUT typu SEU, które prowadzą do innej niż zamierzona implementacji funkcji logicznych. Drugą korzyścią jest możliwość interpretacji tych błędów w szczególnych przypadkach jako stuck-at na wejściach lub wyjściu LUT. Zatem zasoby programowalnych połączeń również są testowane. Stosując ten model, autor rozwinął kompletny system do ewaluacji programów testowych dedykowanych rdzeniom procesorowym implementowanym w FPGA.

Dalsza część tej dysertacji jest poświęcona rozwijaniu metod optymalizacji. Zostały zaproponowane trzy metody optymalizacyjne. Ta część udowadnia, że zestaw wektorów testowych może być znacząco redukowany (dziewięć razy), co prowadzi do redukcji zasobów pamięci i czasu testowania. Autor zaprezentował również podejście do testowania poszczególnych bloków procesora i optymalizację zestawów lokalnych wektorów testowych.

Ostatni rozdział przedstawia problem redundancji logicznych i hardwarowych, które uniemożliwiają uzyskanie 100% pokrycia błędów złożonego systemu implementowanego w FPGA. Rozdział ten przedstawia metody, przykłady, wyniki szczegółowych badań w tym zakresie oraz porównanie metod optymalizacyjnych i rezultatów pokrycia błędów uzyskanych przez różne aplikacje testowe z odniesieniem do bibliografii dziedziny.

1. Introduction

1. FPGA characteristic and applications

Testing of soft-processor cores implemented in SRAM based FPGA circuits. Field Programmable Gate Arrays (FPGAs) are semiconductor devices containing programmable logic components, such as multiplexers, look up tables (LUTs) and programmable interconnections. Logic blocks can be programmed to implement any function of basic logic gates such as AND, OR, XOR, or more complex combinational functions such as decoders or simple arithmetic and logic functions. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complex block of memories.

FPGAs already provide ability of reconfigurability, more performances for Digital Signal Processing (DSP) applications, and finally implemented microprocessor's cores. Consequently FPGAs are increasingly applied to spacecraft electronics for reason of achieving multiple requirements: high performance, low cost of non-recurring engineering, etc. FPGAs are fabricated usually employing SRAM memory cells. FPGAs can change their functions by reprogramming, which is especially useful for low volume devices or for adaptive functions. Thus FPGAs operate often under difficult environmental conditions, such as: on the Earth orbit, spaces with increased radiation, when reprogramming can correct corrupted SRAM-based configuration memory.

2. Motivation

Quick development of integrated circuits technologies, rapid growth of structural and functional complexity of devices cause inseparable need for development of testing methods. Testing FPGAs requires solutions different from those applicable to Application Specified Integrated Circuits (ASICs), **(Teng 2009)**, **(Michinnishi 1997)**, **(Huang 1996)**. Production structural test techniques concentrate on testing individual types of functional blocks **(Abramovici 2000), (Abramovici 1999)** and their interconnections **(Suthar 2006), (Tahoori 2004 A), (Michinishi 1996), (Renovell 2002), (Doumar**

1999). The device is programmed with a number of test configurations and specific test stimuli are applied at each test configuration.

SRAM-based Field Programmable Gate Arrays (FPGAs) are relatively sensitive to Single Event Upsets (SEU), which limits their widespread adoption in safety or mission-critical applications. Single Event Upset can have serious influence on operating FPGAs in space of radiation. SEU occurs when charged particles from the radiation belts or from cosmic rays pass through the silicon and deposit enough energy to induce a fault in the system **(Gaspard 2017)**, **(King 2014)**, **(Wegrzyn 2014 A)**, **(Wegrzyn 2009)**, **(Holbert 2006)**. SEUs play an increasingly important role when technological dimensions of devices decrease and due to more and more complex architectures. Already for dimensions less than 16 nm and very low supply voltages, the rate of random errors produced by neutrons from solar rays would be unacceptable at sea level. The situation becomes worse and worse when altitude increase. Since 45nm other types of circuits as discrete logic (ASIC) are progressively being replaced by FPGAs. Nowadays, microprocessors and memories are implemented in the FPGA matrix **(King 2014)**, **(Xilinx 2019)**, **(Kastensmidt 2006)**. In the last decade design solutions containing MicroBlaze or complex ASIC-FPGA cores as ZYNQ are disseminated. Chips containing one or more ZYNQ, ARM microcontroller cores along Artix or other Ultra SCALE+ FPGA in the same chip are used increasingly. Dimensions of transistors of contemporary FPGAs technology are in the range 45nm as Spartan6, 28nm as Spartan7, Artix7, Kintex7, Virtex7, 20nm – families Ultra SCALE Virtex and Kintex. Finally most modern families Ultra SCALE+ utilizes 16nm and 7nm FinFET technology.

Radiation-hardened FPGAs are often too expensive and either contain usually not enough resources for implementation of more complex designs. Some examples of the radiation-hardened FPGAs are families: PolarFire, ProAsic3 or Fusion Mixed Signal manufactured by Actel company. They are considerable smaller than these offered by Xilinx or Altera (Intel). Some of them are merely proper for CPLD replacement.

Processor cores, which represent one of the basic blocks between FPGA applications, are subjected to SEU-induced faults. In critical applications, an embedded system that is performing its mission should therefore be occasionally tested and reconfigured whenever faults are detected.

3. Purpose and theses of the dissertation

Deeply embedded processor cores are usually hardly accessible from outside. Therefore, testing them is a difficult task, because their inputs are harder to control and their behavior is harder to observe. For the reason of the communication bottleneck between the high-performance Automatic Test Equipment (ATE) and the device under test (DUT), and the limited ATE resources, testing environments are based on Built-in Self-Test (BIST) mechanisms. Professional ATE is usually extremely expensive. The BIST requires designing of additional specialized hardware, developing of advanced optimization algorithms and writing large software applications to build a complete test environment. These very high technologies do not often bring fully satisfying results.

In this work, an attempt of generating compact and efficient, functional test of embedded processor cores implemented in SRAM-based FPGAs is taken up. The solution should be suitable for application oriented BIST. Instruction sequence is composed on the base of data sensitive path principle, thus providing means for randomizing processor operations and consequently increasing the probability of faults detection **(Wegrzyn 2009)**. The developed experiments are targeted at maximal fault coverage, achieved by the developed test program at its as compact as possible architecture. Second important issue is evaluation of this fault coverage. I apply the definition often used in the electronic test domain: *The fault coverage is a ratio of detected faults to all injected faults and is expressed as a percentage.*

Theses

In view of the need for soft-processor's test, and based on the above assumptions, the following theses have been formulated:

Thesis1. *Using sensitive path principle which employs the bijective property of test program may considerably simplify testing procedure and improve fault coverage.*

Thesis2. *Optimization heuristics combined with the proposed fault injection methodology can significantly reduce the number of test vectors required to achieve maximal fault coverage of soft-processors implemented in FPGAs.*

4. Organization of the dissertation

I have divided this work into then chapters devoted to issues of modeling SEU induced faults in FPGAs, testing of FPGAs when such sort of faults occur, analyzing of fault masking mechanisms.

Chapter 2: „Radiation-induced errors in microelectronic circuits." describes the mechanisms for generation of radiation-induced errors in microelectronic circuits. Herein these effects are classified into three types, depending on the extent to which they affect the operation of the FPGA. This chapter introduces the physical mechanisms of Single-Event Upsets (SEUs). Further the propagation of a single-event upset in combinational circuits and flip-flops is illustrated. Single-Event Upset in memory cells and eventually SEUs manifestation in FPGA circuits is considered regarding to basic architectures of FPGA. Different types of SEU induced faults in FPGA and issues related to SEUs modeling in Look-Up-Table (LUT) are introduced in this chapter too.

In **chapter 3: „Software-based self-test of embedded processor cores"** the author introduces the topic related to testing processor cores. A few solutions from bibliography of the subject of structural self-tests of embedded processor cores and functional self-test of embedded processor cores are described herein. In structural self-testing, test-pattern sequences are developed for each processor component, based on the gate-level net list of the individual core components. Since the gate-level details of the processor cores are, in most cases, not available to the designer because their intellectual property is protected. For this reason test patterns are generated in pseudo random way. Alternatively, when the gate-level information of a processor core is available, a deterministic test methodology can be applied, and deterministic test patterns can be generated by an Automatic Test Pattern Generator. During a functional self-test the processor cores are tested by executing a sequence of instructions that exercise the functional behaviour of the processor. The design of this functional self-test is related to the functional description of the processor's instructions.

Chapter 4: „Fault Injection" familiarizes the reader with techniques of Fault Injection (FI) applied for effective evaluation and validation of developing test methods. These methods are classified as simulation based and experimental. Both of them can be hardware based and software-implemented. Often the fault injection constitute serious technical challenge and requires advanced dedicated designing.

Chapter 5: „Proposed Solution: Sensitive-Path Approach" in this chapter the author has proposed the approach, where the goal is to generate a compact test sequence that detects permanent SEU-induced faults of embedded processor cores in SRAM-based FPGAs. The developed experiments are targeted at maximal fault coverage, achieved by developed test program architecture as compact as possible. A new concept of sensitive path of data through the whole program is introduced herein. Test programs for two microprocessors MicroBlaze and PicoBlaze were written. The whole initial idea was creation of a data sensitive path by invention of such an assembler program, which preserves all data. The compact test programs were composed of bijective blocks. Refinements to achieve of full bijectivity are elaborated and described. Experimental results achieved by the test program at various stages of its development are presented in this chapter.

In **Chapter 6 „Reduction of number of test vectors"** The author presents optimization heuristics targeted at reduction of the number of test vectors (Towards bigger processor testing). Meaning of these methods increases considerably in case of more sophisticated processors. Three optimization algorithms were developed: „First the vectors which detect the largest number of faults - Greedy Algorithm", „First the vectors which detect the hardest to detect faults", Hybrid Algorithm that combines features of first and second Algorithms. Also cyclic usage of results are proposed in this chapter. The author has determined optimal sets of global and local test vectors for testing of whole processor hardware and individual functional processor blocks respectively. These experiments were designed to optimize testing of individual blocks when a need arises, and when the other blocks are beyond the interest of the designer e.g. during the design process. These experiments showed significant differences between testability of individual processor blocks.

During work upon the cyclic usage of results method a problem of non-full cycle appeared. Author has solved this problem by application of the Linear Feedback Shift Register. Results of refinement of the PicoBlaze test program using LFSR for all shifts instructions are gathered in this chapter.

Chapter 7: „MicroBlaze case study" describesthe experiments with MicroBlaze processor core chosen by author. The idea of implementation of first test program and the main principles was presented. The data sensitive paths theory is applied, similarly as in case of the PicoBlaze test program composition. Some code examples of initial version of the MicroBlaze test program are explained. Next, problems with evaluation of the test program efficiency are approximated. Despite of these

problems, certain results of researches upon this efficiency, which can be compared with the bibliography of subject are presented. Finally results of researches on PicoBlaze are applied to composition of the MicroBlaze test program and examples of code are presented at the end of this chapter.

Chapter 8: „Evaluation of the test program" presents evaluation schemes of test program. This chapter discusses such topics as: environment composition for experiments, proposed technique and environment for dedicated FPGAs fault injection. It also describes PicoBlaze structural VHDL, and auxiliary scripts.

Chapter 9: „Problem of faults masking" analyses in details problem of faults masking. The problem of faults masking is considered due to every block of the microprocessor. Herein undetected faults remained after testing program execution are classified into a few categories. Different kinds of logic and HW redundancies are investigated. Certain mechanisms of fault detection are explained on examples. Methods to solve the fault masking problem are proposed in some cases.

5. Thanks

Thank you to my supervisor D.Sc. Ernest Jamro, Prof. TU AGH for invaluable help; advices at improvement of researches, guidance and feedback throughout this dissertation. Thanks also to co-supervisor PhD. Agnieszka Boruch-Dabrowska for fruitful discussions and advices.

2. Radiation-induced errors in microelectronic circuits

This chapter describes in more detail the mechanisms for the generation of radiation-induced errors in microelectronic circuits. The effects are classified into three types, depending on the extent to which they affect the operation of the FPGA.

2.1. Single Event Effects

Radiation from space can cause major errors in integrated circuits. A so-called single-event effect (SEE) occurs when charged particles pass through the silicon and emit some of their energy. Such particles can be classified into two main types:

- charged particles (e.g. electrons, protons, and heavy ions),
- photons of electromagnetic radiation (e.g. x-rays, gamma rays, and ultraviolet rays).

These particles mainly originate from the Van Allen belt, although the heavy ions have their origin in solar flares, the magnetosphere and cosmic rays. When the high energy particles pass through the integrated circuit they induce the phenomena of ionization and excitation inside the semiconductor.

SEEs are classified into three types in reference **(Gaspard 2017)**, **(King 2014)**, **(Holbert 2006)**, based on the amount of energy lost by the charged particles in the devices:

- a single-event upset (a soft error),
- a single-event latch up (a soft or hard error),
- a single-event burnout (a hard failure),
- a single-event transients (a soft error).

A single-event upset (SEU) is a radiation-induced error in a microelectronic circuit. The phenomenon of a SEU occurs when the charged particles lose their energy by ionizing the semiconductor which they pass through. SEUs are classified in the first category, as soft, non-destructive errors. They manifest themselves in different, unexpected operations of the circuit, and in order to restore the correct operation of the circuit it is only necessary to reset it or reprogram it. Semiconductor devices, such as MOS transistors, BJTs, resistors, capacitors, optical devices, etc. and hence both analogue and digital circuits are sensitive to SEUs. In practice, SEUs manifest themselves as bits-flop in a sequential part of a system and as transient pulses in logic circuits. Sometimes, it is possible for one ion to cause multiple errors on more than one bit, which is, as a result, in most cases easier to detect. Multiple SEUs occur, for example, in SRAM-based FPGAs, which leads to the incorrect operation of an application implemented in the FPGA.

A single-event latch up (SEL) is a radiation-induced error that can damage a device. A SEL phenomenon manifests itself as an increase in the operating current above an acceptable threshold. A SEL can decrease the supply voltage or overheat a device, which can then result in damage to the device and the power supply. This phenomenon is usually caused by heavy ions. Only in devices with very small dimensions protons can cause a SEL. It is possible to power-off and then power-on a device in order to prevent the catastrophic results occurring as a result of a SEL.

A single-event burnout (SEB) is a more powerful phenomenon then a SEL and its manifestation is mostly critical in power MOSFET transistors. A SEB manifests itself as a current increase and leads to the destruction of the device. A SEB causes the burnout of power transistors, gate rupture, and as result the freezing of bits. Moreover, a SEB can cause the device to switch itself on. These phenomena, which have been known since the 1980s, tend to occur at low temperatures. At higher temperatures they occur less frequently. The phenomenon of a single-event gate rupture (SEGR) occurs in power MOSFETs. It consists of the breakdown of a gate insulator and conduction through a layer of the gate insulator, which subsequently leads to destructive burnout. SEGRs were also observed in other semiconductor devices, such as bipolar junction transistors.

A Single Event Transients (SET) can appear in combinational logic or may be latched into memory or a flip-flop. According to **(Gaspard 2017)** deposited charge as result of ionization, can be collected by the transistor's source and drain junctions. In a case of off - state transistor, the node voltage can be temporarily changed. This results in a Single Event Transient (SET). Such a phenomenon is possible if deposited by ionization charge is enough to recharge the associated with the transistor node capacitance. Then the SET amplitude can swing from rail to rail or wire to wire. The SET pulse width

is proportional to the charge collected by the off-state transistor, the transistor node capacitance, and value of restoring current, which drives the transistor.

2.2. The physical basis of a SEU

For practical reasons I take into consideration single-event upsets (SEUs), because they are non-destructive and there are ways of alleviating the problems caused by them. We can distinguish two main physical mechanisms that lead to the occurrence of a SEU **(Gaspard 2017)**, **(King 2014)**, **(Holbert 2006), (Ohlsson 2002):**

- ionization, induced by the heavy ions from cosmic rays and solar radiation. The ionization mechanism is illustrated in Figure 2.2.1,
- complex nuclear reaction, leading to spallation induced by the high-energy protons.

Spallation is a nuclear reaction that involves the ejection of particles from the nucleus. It occurs naturally in earth's atmosphere, owing to the impacts of cosmic rays. Spallation is the process in which a heavy nucleus emits a large number of nucleons as a result of being hit by a high-energy particle. Usually, it is heavy nuclei ions, like ^{25}Mg, that have the ability to induce a SEU by spallation. Other particles that also have such an ability are Si(n, α)Mg, Si(n, p)Al, Si(p, 2pAl), Si(p, p, α). The mechanism of spallation is illustrated in Figure 2.2.2.

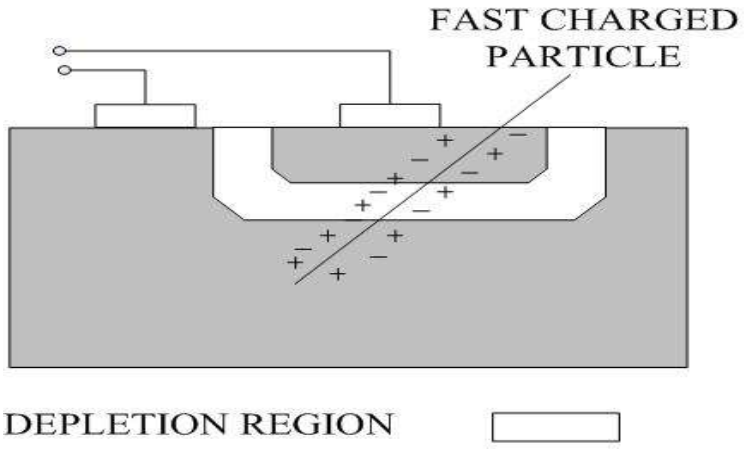


Figure 2.2.1: Radiation-induced errors in microelectronic circuits: Ionization

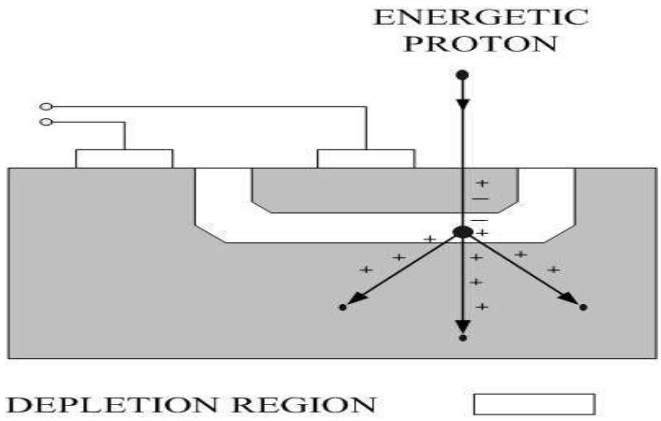


Figure 2.2.2: Radiation-induced errors in microelectronic circuits: Spallation

SEU induced errors are critical in space applications and have been subject of intensive studies. Recent researches have revealed that 90% of all SEUs in interplanetary space are induced by protons. This fact emphasizes the significance of the phenomena induced by protons, as opposed to the initial assumption that SEUs would be induced mainly by cosmic rays. The experiments reported in **(Holbert 2006)** were carried out in space, above the South Atlantic.

More recent physical experiments were led by NASA, and were reported in paper **(Megan 2012)**. These researches upon susceptibility of electronic elements and IC to cosmic radiation proceeded on a deck of space aircraft during space missions. Electronic elements have been ionized by high-energy electrons trapped within the Jovian radiation belts. These electrons have energy and mission flounce orders of magnitude higher than observed in the Earth's trapped radiation belts. For instance a popular P JFET transistor 2N5116 was exposed to radiation. Four elements were irradiated with gammas, two were irradiated with electrons, and two were used as controls. Results of the experiment are presented in Figure 2.2.3 **(Megan 2012)** bellow. Diagrams in this figures are an average of four parts irradiated by gamma rays, two parts irradiated by electrons, and two control parts. It is visible, that the rate of the magnitude changes of the parts irradiated with electrons is faster than the magnitude of the parts irradiated with gamma rays. Degradation Level is expressed in [krad(Si)]. - unit of absorbed radiation dose [1 krad = 10 J/kg]. The error bars indicate one standard deviation.

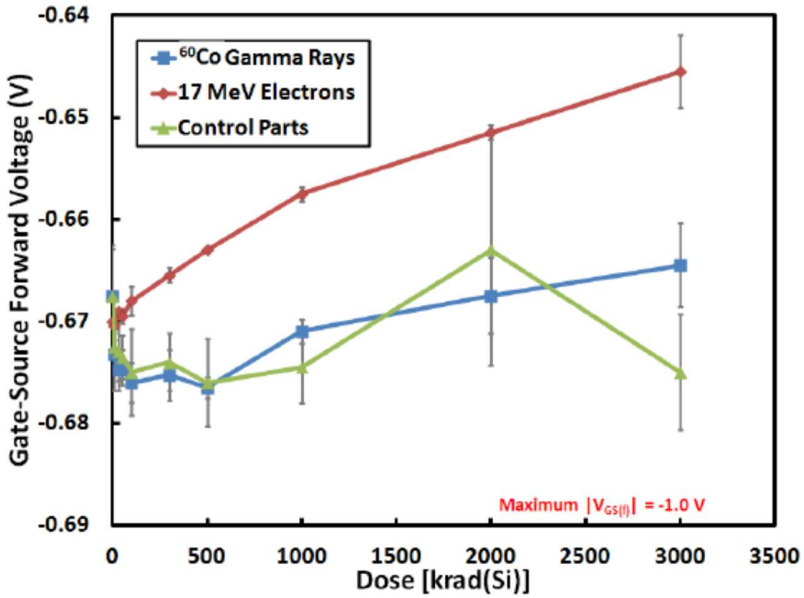


Figure 2.2.3: Changes of magnitude of the Gate-Source voltage of JFET transistor irradiated with electrons and gamma rays

2.2.1. Critical charge

The particles from the radiation disturb the balance of the electric field inside a semiconductor device by generating a large number of free electron–hole pairs in a bipolar transistor. As a result, a large electric field exists across a reverse-biased p-n junction. The free carriers that appear as a consequence are collected by this field, which results in the generation of a transient noise pulse, which can then generate an SEU in flip-flops **(Kastensmidt 2006)**. The charge-deposition mechanism is often represented by charge-deposition waveforms. These waveforms are different for different radiation sources, depending on their incoming angle, which sets the technological parameters, i.e. the doping profile.

According **(Gaspard 2017)**, **(King 2014)**, the critical charge (Qcrit) is the minimum amount of charge that must be collected to result in a SEU. Qcrit of a flip-flop decreases if decrease node capacitance, supply voltage or current which drives transistor. Because CMOS technology feature sizes continuously decrease, which follows supply voltages, node capacitance and driving current, this results in decreasing Qcrit. Estimation of critical charge as a function of technology node feature size is presented in Figure 2.2.4:

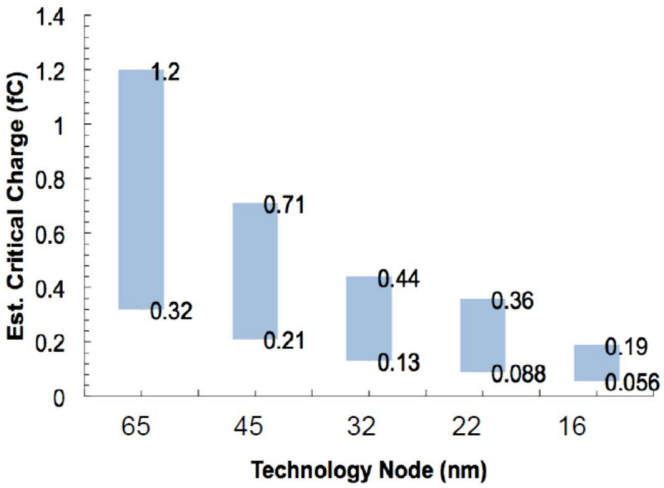
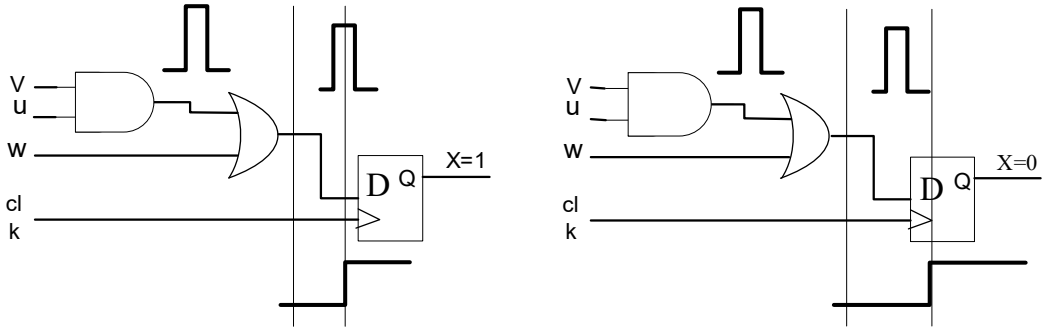


Figure 2.2.4: Estimation of critical charge as a function of technology node feature size

2.2.2. The propagation of single-event upsets in combinational circuits

A particle from a cosmic ray can cause a glitch in the output voltage of a logic gate in combinational logic. This so-called single-event transient (SET) can be propagated in a combinational circuit **(King 2014)**, (**Hellebrand 2007)**, where it acts like a single-event upset (SEU). These phenomena occur when the propagation paths are sensitized in the logic, and the glitch arrives at the flip flop during a latch window. The simple example in Figure 2.2.5 below explains the propagation of the glitch.



a) SEU error occurs b) SEU error does not occur

Figure 2.2.5: Radiation-induced errors in microelectronic circuits. SEU in a combinational circuit

If a particle hits the AND gate and produces a glitch at the output, it can be propagated exclusively through the OR gate for w = 0 as is shown in Figure 2.2.5a. If the glitch at the end of the OR gate occurs before the next rising edge of the clock, it cannot be propagated as presented in Figure 2.2.5b.

The single-transient fault (STF) is defined in reference **(Hayes 2007)** by Hayes et al. They assume a circuit with a number of logic lines equal to *k.* This circuit possesses a certain number of inputs and outputs. Moreover, the circuit is described by the set of internal states, the next state's function and the output state's function. A single transient fault in the circuit causes line l to be stuck at zero or stuck at one for a single clock cycle **(Bushnell 2000)**. This leads to an interesting question: what is the probability of an SEU producing an erroneous output from the circuit within a certain number of cycles after the fault has occurred? The authors assume that the faults appear and disappear within a single clock cycle. However, it is possible that the faults tend to occur at random times and are likely to affect all the states of the circuit. Let us assume that a circuit possesses *k* lines, *n* primary inputs, and a single primary output *y*, and assume that the probability of each STF is the same. The probability of STF is defined as the total number of possible errors produced at *y* by STFs, divided by the total number of possible STFs **(Hayes 2007):**

$$P_{err}(z) = (\sum_{l} \text{No. of tests for the faulty line l}) / \text{k } 2^{n+1}$$

If the circuit has *n* inputs it means an „elementary" gate G of the N (AND) or (N) OR type, and then the above equation takes the form:

$$P_{err}(z) = (\text{n}+2^{n-1}) / (\text{n}+1)2^{n+1}$$

The example of logical functor is presented in Figure 2.2.6.

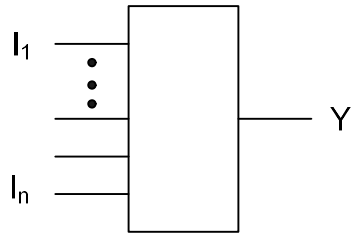


Figure 2.2.6: Radiation-induced errors in microelectronic circuits. Example of logical functor

The analysis of erroneous behaviour is a very complex process, because it depends on many physical factors **(Maheshwari 2004)**. For this reason the research in this area requires the use of electrical and probabilistic models that are technology- or application dependent.

For practical reasons I consider an example that makes clearer the mechanism of the incorrect operation of logical gates. For instance, I consider the three-input CMOS gate NAND3 depicted in Figure 2.2.7.

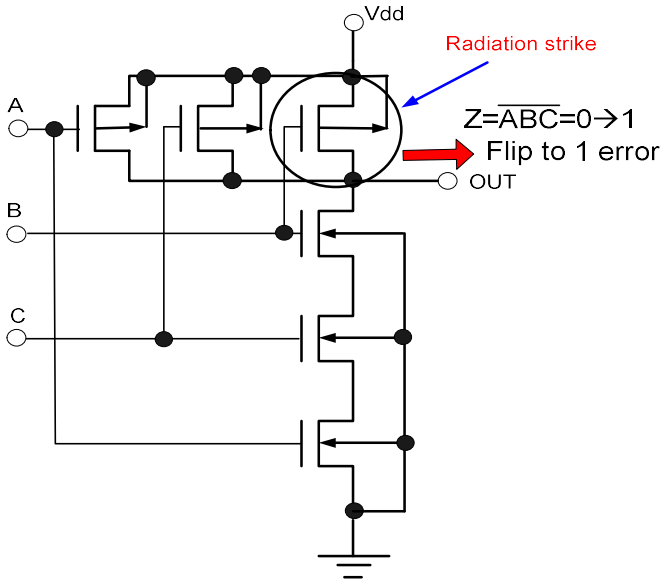


Figure 2.2.7: Radiation-induced errors in microelectronic circuits. Transient flip-flop error in a NAND gate

There is a probability that charged particles will strike one or more transistors of a gate. A radiation strike can upset one or more of its transistors, causing the output *Z* to undergo a transient flip-to-0 or flip-to-1 error. The specific error depends in part on the input pattern ABC when the strike occurs. The input ABC = 111 flips *Z* from 0 to 1 if one of the gate's p-transistors is upset, as is the case in Figure 2.2.7. The strike can cause flip-to-1. Inversely, when the output Z = 1, under the input patterns 110, 101 and 011 only one n-transistor (i.e. the one with logic 0 at the gate) is susceptible to the strike. Similarly, with the input equal to 000, all the three *n*-transistors must be upset to produce an output bit flip-to-0.

2.2.3. The propagation of single-event upsets in flip-flops

Flip-flops are usually used for temporary storage of data between operations inside a processor. As an especially visible example can serve here pipelining processor architecture. Single Event Transient (SET) can manifest itself in two ways **(Gaspard 2017)**. First of them is when a SET occurs within the latch, and second if SET is propagated from logic circuit and can be latched into the flip-flop at a clock edge. In this way a SET can create a SEU. This occurs if the SET continues longer than the feedback loop delay of the flip-flop latch. Many architectures of flip-flops reminds SRAM cells, because they uses often a feedback loop similar as inside SRAM. For the reason, that some flip-flop latches are almost identical to SRAM cells, mechanisms that cause SEUs in flip-flops is similar to this in SRAM cells. Figure 2.2.8 **(Gaspard 2017)** presents mechanism of generation SEUs inside a latch.

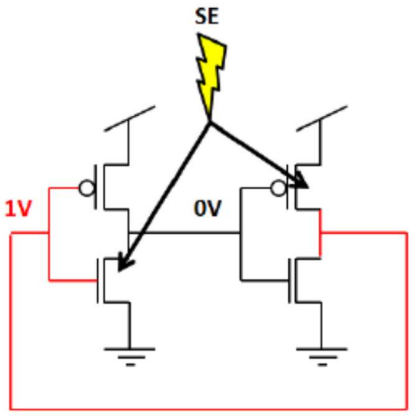


Figure 2.2.8: Mechanism of generation SEUs inside a latch

2.2.4. Single-event upset in memory cells

A transient current pulse is generated if a charged particle strikes one of the sensitive nodes of a memory cell, such as the drain in an off-state transistor. Such a current pulse can turn on the gate of the opposite transistor. This Single Event Transient current pulse can result in an SEU if the pulse width is longer than the feedback loop delay of the design **(Gaspard 2017)**. Memory cells have two stable states: one that represents a stored „0” and one that represents a stored „1”. In each state, two transistors are turned on and two transistors are turned off. A bit-flop in the memory element occurs

when an energetic particle strikes the drain. This event can produce an inversion in the stored value, which means a bit flip in the memory cell. This effect is called a single-event upset (SEU). The classic architecture of a SRAM memory cell with the area sensitive to particle strikes marked is illustrated in Figure 2.2.9, **(Gaspard 2017)**, **(Kastensmidt, 2006)**.

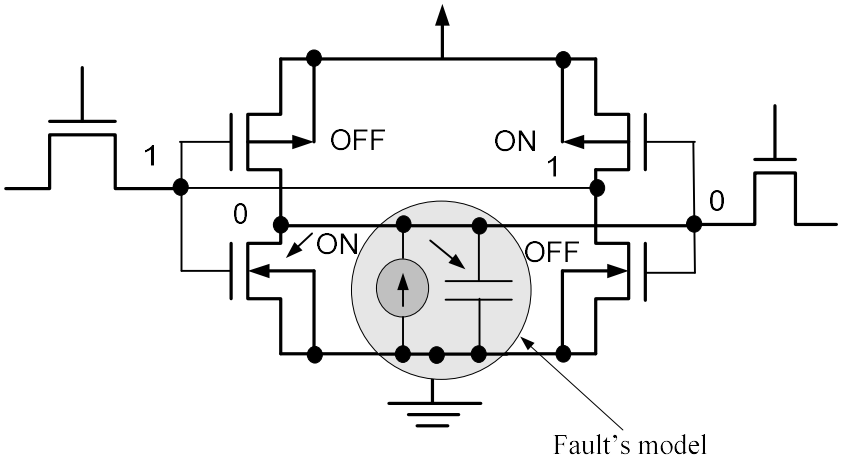


Figure 2.2.9: Radiation-induced errors in microelectronic circuits. Single Event Upset (SEU) effectin a SRAM Memory cell

2.3. SEU manifestation in FPGA circuits

The susceptibility of current technologies to SEUs ranges from CMOS/SOS (the least susceptible), to CMOS, to standard bipolar, to low-power Schottky bipolar, and then to NMOS DRAMs (the most susceptible). A long list of papers has been published on the subject of SEU-induced errors in microelectronic circuits, among them **(Katz 1997)**, **(Katz 1999)**, **(Karp 1993)**, focusing on the radiation effects related to current field-programmable technologies.

SRAM-based field-programmable gate arrays (FPGAs), which are nowadays massively used in different embedded-system applications, are relatively sensitive to single-event upsets (SEUs),

(Xilinx 2013). The testing of a FPGA is a difficult and important problem. The testing strategy and the associated procedures depend on the target fault types and the operating modes (off-line test, on-line test, concurrent test), **(Abramovici 1999)**, **(Abramovici 2000)**. In the following I briefly describe the most common types of faults in SRAM-based FPGAs, and in the next chapter, I review the methods for testing FPGA circuits. The typical architecture of the FPGA circuit is presented in Figure 2.3.1.

The edges of the chip-FPGA are surrounded by programmable I/O block (IOB) resources. The blocks of RAM are near the edges of the chip. At the centre of the chip is a two-dimensional array of configurable logic blocks (CLBs). A CLB consists of a certain number of slices, and each slice contains a certain number of look-up tables (LUTs), flip-flops, carry and routing logic. Figure 2.3.1. presents a simplified top-level overview of the architecture.

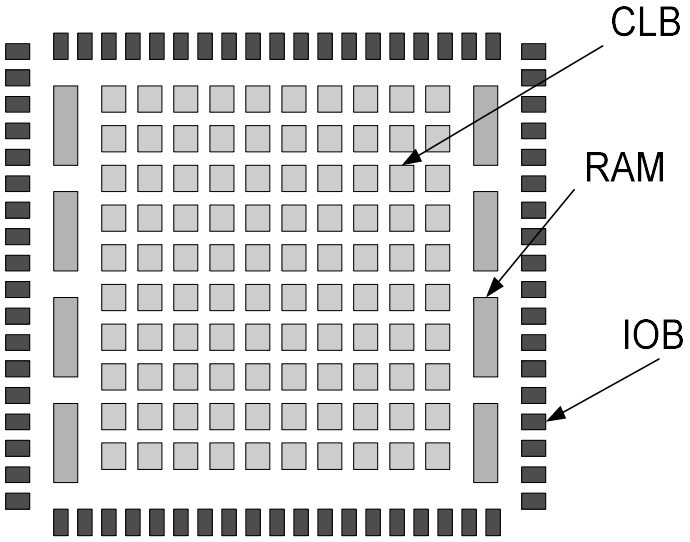


Figure 2.3.1: Radiation-induced errors in microelectronic circuits. Simplified block scheme of a FPGA

The routing of the signals for the CLB array is ensured by wires that connect the CLBs. However, although there are a lot of different architectures, generally a FPGA consists of an N×N array of configurable logic blocs (CLB) and programmable I/O blocks. Between these blocks, there are programmable interconnections which are provided by a single-switch box (SSB).

The programmable interconnections are provided by a single-switch box (SSB). A SSB consists of a matrix of programmable interconnect points (PIPs), with each PIP being a pass transistor that can connect two wire segments.

As described in **(Rebaudengo 2002 A)**, the internal architecture of the configurable logic block is shown in Figure 2.3.2, below:

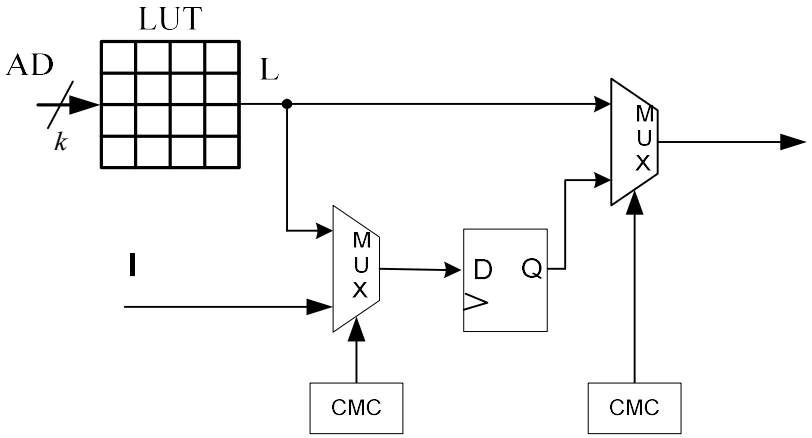


Figure 2.3.2: Radiation-induced errors in microelectronic circuits. SEU resources inside CLB

The basic internal architecture of a CLB, shown in Figure 2.3.2, is built by three components: a look-up table (LUT), multiplexers and D flip-flop. A LUT can be programmed to implement any k-input combinational function. The CMC box in Figure 2.3.2 represents configuration memory cells. Depending on the value applied to the input lines, the table selects a CMC addressed by the input pattern, and the cell's output provides the function's value. A LUT can therefore implement any of 2n functions of its n input, where $n \leq k$, k is LUT address bus bit width. In programming the FPGA the CMCs corresponding to the LUTs are loaded with the bit pattern corresponding to the function truth table. In the CLB, the connections among the input and output lines, the LUTs, and D flip-flops are controlled by CMCs. The interconnection structure, surrounding the CLB, is composed of connections configured as pass transistors, also controlled by a CMC.

In practice a typical logical unit has a more complex architecture. For example, the Xilinx Virtex 5+^{TM} series has a CLB consisting of two slices. Every slice possesses four six-input LUTs, carry logic, eight flips-flops, wide-function multiplexers.

2.3.1. Fault model in FPGA

In earlier references **(Renovell 2000 A), (Tahoori 2004 B)** only an approximate model with possible faults affecting the configuration memory is described. However, this approximation does not consider the faults affecting the values of the memory cells composing each LUT. It takes into consideration only the faults affecting the output value of the LUT.

Another approach **(Rebaudengo 2002 A)** proposes a more accurate fault model to test. This approach takes into account the stuck-at faults affecting the memory bits composing the LUTs, and the coverage of the defects affecting the memory bits composing the LUTs is improved. The model analyses more accurately the functional effects induced by the faults affecting the logic elements in a logical unit (see Figure 2.3.3):

- the LUT's address lines (AD),
- the inputs and outputs of the LUT's memory cells (LUT),
- the LUT's output (L),
- the data input (I),
- flip-flop input (D),
- flip-flop output (Q),
- multiplexer inputs (M1(0), M1(1), M2(0), M2(1)),
- multiplexer outputs (M1, M2),
- multiplexer control signals (M1A, M2A),
- CLB output (F).

All these faults are stuck-at-0 and stuck-at-1. It is important to emphasize that the faults which belong to the same net are equivalent. In the example of Xilinx Virtex FPGAs, we can define the classes of equivalence composed of the following signals:

- I and M1(0),
- M1 and D,
- Q and M2(1),
- M2 and F.

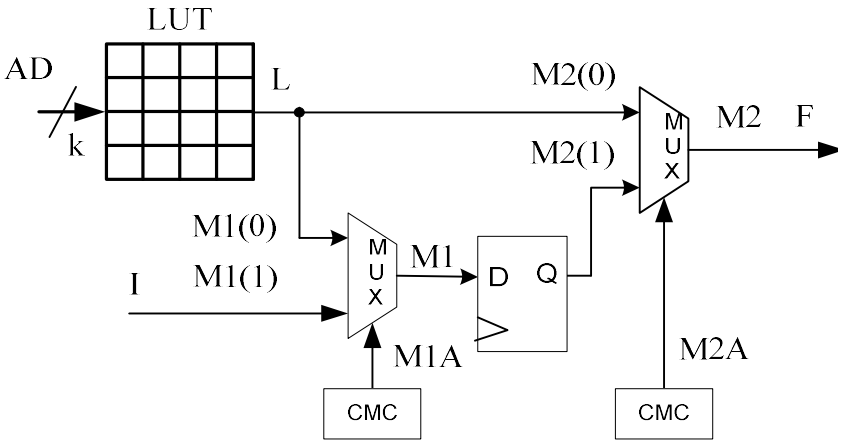


Figure 2.3.3: Radiation-induced errors in microelectronic circuits. Fault model in a FPGA

For example, if the multiplexer configuration inputs and the LUT memory-cell inputs are set to constant values their value cannot be modified during the test. This results in a redundancy of faults in these resources. These faults may be removed from the fault list.

As a contrasting example I can demonstrate that the relevant stuck-at faults alter the value of the LUT bit cells which cannot be classified as being equivalent to the fault affecting the LUT's output. For instance, I can take into consideration a LUT implementing the function, F = AB + C where A, B and C are the inputs of the function. This function is described by the truth table 2.3.1.

The stuck-at-1 fault alters the value of the memory-bit cell stored at the address A = 0, B = 1 and C = 1. This modifies the LUT function into F' = AB + (~B)C, which is not equivalent to a stuck-at-1 or stuck-at-0 fault affecting the LUT output and the elimination of the above fault leads to a shortage in the range of possible faults. In contrast, the stuck-at-0 alters the value of the memory-bit cell stored at the address A = 0, B = 0 and C = 0. However, this does not modify the function and thus it is a redundant fault.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 2.3.1: Radiation-induced errors in microelectronic circuits. An example of a LUT with SEU induced error for A = 0, B = 1, C = 1

A radiation-induced error may affect the application configuration, which may lead to a faulty application operation. A SEU may alter the content of a configurable logic block or produce modifications in the interconnections, thus giving rise to totally different circuits from those intended. SEUs may modify the memory elements the design embeds as the content of a register in the data path, or the content of the state register in a control unit. Accordingly, the bibliography of the fault effects can be classified in the following classes:

- effect less: the output behavior of the faulty circuit is the same as the fault-free system,
- malfunction: the output behavior of the faulty circuit differs from that of the fault-free circuit.

2.3.2. Types of errors induced by SEU in FPGA circuits

SEU-induced faults can be categorized according to their location into two broad classes **(Rebaudengo 2002 B), (Suthar 2006), (Renovell 1997), (Renovell 1998), (Syam 2005):**

- Inter-CLB resources: a fault modifies the routing of the signals between two or more presented in Figure 2.3.4. This results in the cutting off the existing connections between the CLBs before the manifestation of the fault. Moreover, additional connections between CLBs may originate as a result of the appearance of a SEU.

- Intra-CLB resources: a fault modifies the configuration of the resources inside a single CLB of the FPGA. On the basis of the architecture presented in Figure 2.3.5 we can classify the resources that can be altered in two classes:

1. Routing: the SEU modifies one of the configuration bits of the multiplexer that the CLB embeds (resources A, B and C in Figure 2.3.5),

2. Look-up tables: the SEU modifies one bit in the look-up table that the CLB embeds (resources LUT in Figure 2.3.5).

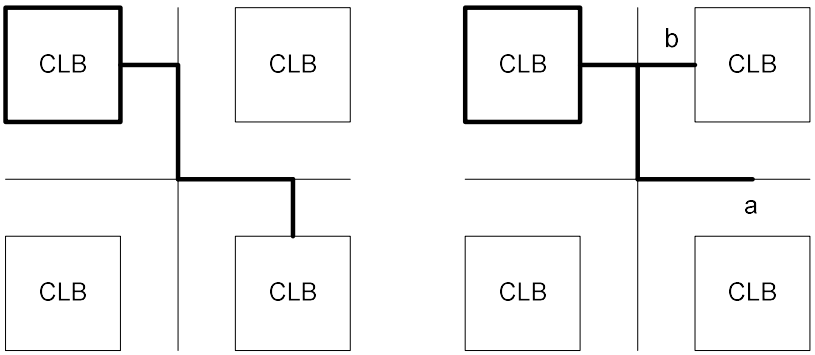


Figure 2.3.4: Radiation-induced errors in microelectronic circuits. Routing SEU resources

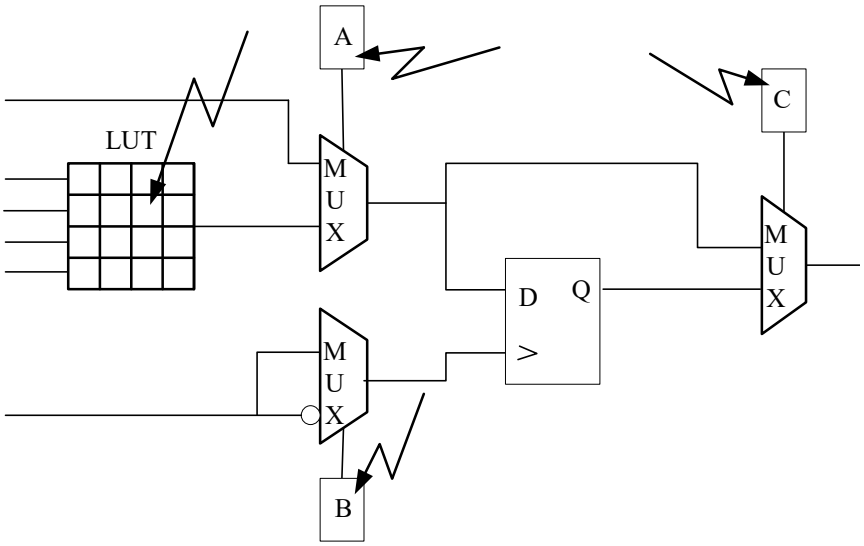


Figure 2.3.5: Radiation-induced errors in microelectronic circuits. Intra-CLB resources

3. Software-based self-test of embedded processor cores

Complex systems-on-chips (SoC) are usually composed of many embedded processor cores. A processor core downloaded into FPGAs-based designs possesses several advantages, for example, the reuse of the chip, adopting the number of cores to a specific task and the possibility of implementing a complex design exclusively by programming. Hence, the cores can be applied in various applications and this then lowers the cost **(Batcher 1999)**, **(Chen 2000).**

Testing a core that is deeply embedded and has poor accessibility is a difficult task. Embedded processor cores are difficult to test because their inputs are harder to control and their behaviour is harder to observe. For this reason, the communication bottleneck between the high-performance Automatic Test Equipment (ATE) and the device under test (DUT), and the limited ATE resources, lead to solutions based on built-in self-test mechanisms. With this approach, both the test-pattern generation and the evaluation of the test results are performed by the processor under test, i.e. Built-in-Self-Test (BIST) methodologies. However, BIST methodologies rely on a scan to deliver the test patterns and are often effective enough, but they cannot be applied to systems containing embedded processors. The approach to Built-in Self-Test is described with practical examples from bibliography **(Renovell 2000 C), (Renovell 2001).** Since FPGA circuit resources are not normally 100% occupied by the design, the defects located in some areas of the chip that are not used by a particular design may be tolerated. These problems are described in chapter 3.2.

There are two main approaches to software-based self-testing: the structural approach and the functional approach.

3.1. Structural self-test of embedded processor cores

In structural self-testing, test-pattern sequences are developed for each processor component, based on the gate-level net-list of the individual core components. Since the gate-level details of the processor cores are, in most cases, not available to the designer because their intellectual property is protected, there are serious restrictions when it comes to practical applications. A high-level structural self-test methodology **(Kranitis 2002 A)** tries to overcome this problem, based on knowledge of the Instruction Set Architecture (ISA) of the processor and its Register Transfer (RT) level description. Additionally, the gap between the operating frequencies of the ATE and the operating frequencies of the System on Chip (SoC) can lead to a large number of undetected faults. Knowledge of ISA is indispensable for the developed methodology. In the considered case of MicroBlaze microcontroller, I use ISA moreover for PicoBlaze, I gathered knowledge about structure of implemented hardware.

In the structural testing methodology, pseudo-random pattern sequences are developed for each processor component before the test. In the test-execution phase, the pseudo-random test patterns are usually expanded on-chip by a software-emulated Linear Feedback Shift Register (LFSR) and then stored in the embedded memory. Then, the test patterns are applied by software-test-application programs and the responses are collected in the memory again. The gate-level details of the processor architecture are required for this methodology **(Renovell 2000 C)**, **(Renovell 2001)**. The instruction set imposes a constrained test generation of the deeply embedded functional modules of the processor. This task consumes an excessive amount of time and may sometimes lead to unacceptable low level of fault coverage. Additionally, the pseudo-random test methodology leads to a large self-test code. I applied the software-emulated Linear Feedback Shift Register (LFSR) to test pattern generation, but in deterministic way and exclusively for specified blocks of a processor core, where this actually has given a good effect, as will be showed in chapter 6.3.1. This application of the LFSR differs from typical applications.

The paper **(Kranitis 2002B)** presents Instruction-Based Self-Testing of Processor Cores. The methodology consists of three steps:

- **information extraction** from the processor's instruction-set architecture and the RT level description. In this step the effects of the execution of each instruction, for every component are extracted,

- **instruction selection -** for every component the set of operations is selected. Identified (decoded) are instructions which given component performs,
- **operand selection -** in this step the deterministic operands are considered, that must be applied to each component to achieve high structural fault coverage.

Alternatively, when the gate-level description of a processor core is available, a deterministic test methodology can be applied, and deterministic test patterns can be generated by an Automatic Test Pattern Generator. This method is efficient only when the number of test patterns is low. The Register Transfer (RT) level description represents the connections among the functional parts of the processor, such as Arithmetic Logic Unit (ALU), multiplexers, and shifters, and the storage elements, such as registers, flags and steering logic modules such as bus elements.

To implement a high-level structural self-test program, there is no need for a synthesis and gate-level description of the tested components, which is its important advantage. On the contrary for a deterministic methodology, a basic knowledge of the functionality and the functional blocks inside the component is required. This information can be easily obtained from the control signal applied to the block or a related output. The method makes possible a significant reduction in the number of processor instructions, the program size (the number of bytes that have to be downloaded by an external ATE to the memory or stored in the ROM) and the response data size (the number of test response bytes that are stored in the memory and later uploaded by an external ATE or compressed by a test-response-analysis program).

3.2. Functional self-test of embedded processor cores

During a functional self-test the processor cores are tested by executing a sequence of instructions that exercise the functional behaviour of the processor. The design of this functional self-test is related to the functional description of the processor's instructions. In earlier implementations, individual instructions were tested with a set of deterministic test patterns and the results were compared with stored reference values.

Functional testing is an effective solution that can overcome the limitations mentioned at the beginning of this chapter. The solution involves forcing the microprocessor to execute a specific test program. The testing of microprocessor cores is a challenging task. Hence, new testing methods are being developed, such as at-speed testing, an automatic test generating program, which exploits e.g. an evolutionary algorithm.

I defined a new model of faults dedicated for FPGAs **(Wegrzyn 2009)**. The model is Single Event Upsets (SEUs) based, and consist in changes of logical functions implemented by Look Up Tables (LUTs) in FPGA in case of SEU fault occurrence. The mechanism of fault generation is detailed described in chapter 2 and analysis of the influence of faults on operation of specified blocks and assembler instructions is described in chapter 8.3.

The at-speed testing of microprocessors with external testers becomes increasingly difficult as the frequency increases **(Batcher 1999)**, **(Chen 2001)**. A proper solution to the problem is a built-in self-test (BIST). This has been developed over the past several years. In his earlier work **(Batcher 1999)**, developed methods for the efficient compilation of self-test programs for embedded processors. However, these methods do not make possible the generation of self-test programs for the test engineers. Also, Shen and Batcher **(Batcher 1999)** have attempted to develop the functional self-testing of processors. Their approach consists of generating and applying random-instruction sequences to the processor core. I also carried out such experiments with instructions of PicoBlaze processor, and I evaluated fault coverage of such a test program. These effects were disappointing. However this approach may be classified as supplement to deterministic self-instruction program.

Built-in-self-test methods **(Batcher 1999)** consist of a scan, to deliver the test patterns, which are effective enough. This may often be difficult to apply to systems containing embedded processors. Access to the embedded core for scan insertion can be difficult. The processor architecture itself may make it impossible to use scan test methods, even the design for test rules for such methods requires the design of test rules. The structure of the microprocessor is usually complicated (flip-flops, asynchronous logic, internal three state, gated clocks), and this structure can cause a lot of problems for scan-based testing.

The Instruction Randomization Self-Test (IRST) **(Batcher 1999)** methodology utilizes the functional behaviour of a microprocessor and a stuck-at fault model to obtain high fault coverage. The test is performed by continuously executing a random stream of processor instructions and compressing the execution results using internal test hardware. The advantage of the method is that it can be applied to various processor architectures with the following features:

- the processor must fetch instructions from a read/write memory,
- the instructions must perform operations on an internal register and/or memory,
- a single instruction cycle is required,
- the processor must be able to execute a branch instruction.

The disadvantage of functional testing is low fault coverage, for the reason that it does not consider the RTL structure and the excessive power consumption in the BIST mode. This can be reduced by suitable techniques, such as scheduling. The fault coverage can be improved with techniques of deterministic BIST. The BIST overhead is usually reduced by generating pseudo-random test patterns, using circuits as accumulators, embedded processors, sequential circuits, etc. In scan-based BIST, the test overhead is usually reduced by a partial scan **(Chen 2001)**. The circuit under test is driven by random test patterns and using some non-functional mode may cause a problem with the bus connections **(Chen 2001)**. Moreover, functional testing that uses some RTL information is not exactly correct for embedded microprocessor cores **(Batcher 1999)**. The testing methods used for the fault validation utilize gate-level information. They can be applied in industry for processor testing, which is their main advantage. There is a difference between these methods and the Instruction Randomization Self Test (IRST), because the IRST is executed at-speed and its algorithms utilize hardware information. The instruction randomization is performed with dedicated hardware, which modifies certain instruction fields, and in this way the instruction remains full of meaning and its operand gets randomly permuted. In this way the operation of the processing is explored in a larger number of situations, which increases the fault coverage. The IRST solution is based on hardware and software. The functions of the hardware are as follows:

- to modify the test software to provide a pseudo-random sequence of instructions,
- to monitor the instruction fetch and R/W activity,
- to provide a source of randomized seed data, which the test software uses to randomize the register operands.

The test hardware consists of a randomizer, modifiable instruction storage (MIS) memory, a fixed test-instruction (FTI) memory and a Test Response Compression Device (TRCD) **(Batcher 1999)**.

The test software program is highly optimized for detecting faults. There are many techniques for increasing the controllability and the observability of the test software. The initial test program is important for the controllability of the MIS software. The main task of the software program

is to provide a source of random stimulus for control over various data and the control path in the processor core. Below is the overview of several test approaches.

The solution **(Chen 2000)**, **(Chen 2001)** consists of two steps. The pre-test step is the generation of realizable component tests and the encapsulation of component tests in self-test signatures. In this step the tests are developed for individual components of the processor, such as ALU, PC, etc. The faults are injected into the structure during the component test generation. There are two types of component tests: random tests and deterministic tests. For the random tests, the test for each component is characterized by the self-test signature. The seed and pseudo-random generators are used, and so-called self-test signatures are loaded into the processor memory before the test instead of during the actual tests. The signature can be expanded on-chip into test sets using a pseudorandom number of generation programs. For the deterministic tests, the tests are loaded directly into the processor memory before the test. The use of self-test signatures reduces the loading time for loading the test sets into the memory for storing all the test patterns at the same time.

Application of the signature analysis seems to be one of possible solutions to testing more sophisticated processors and their components as my case study of MicroBlaze/SecretBlaze. (There is a huge number of 2 to the power of 32 of possible test vectors, what is considered in the next chapters). Whereas in case of small eight-bit processors I am able to lead exhaustive tests using all 256 test vectors.

The approach uses a software tester embedded in the processor memory. The software tester consists of a group of programs for the test generation and the test application. The main feature of the solution is that the instructions in the software tester are carefully chosen in order to deliver the previously prepared structural tests.

In the self-testing, the step of applying on-chip tests and the response collection use the functionality of the processor under test. At the component level a structural faults model is applied. At the processor level the method uses the functionality of the processor for the structural tests of each component at-speed. In my case I have to consider implementation of processor inside FPGA additionally.

It is impossible to deliver some test patterns when the delivery of the component tests consists of the functionality of the circuit. Thus, component tests are constrained by the processor's instruction set, the validity of the input values, etc. When the component tests are developed under the constraints imposed by the processor's instruction set, they can be generated by delivery programs to apply the component tests **(Chen 2000)**. Figure 3.2.1 bellow explains the self-test methodology.

When self-test signatures are used, an on-chip test-generation program emulates a pseudo-random pattern generator and expands the signatures into test patterns. The test patterns are applied to components by a test-pattern delivery program at the speed of the processor, which also collects the tests and saves them to the memory.

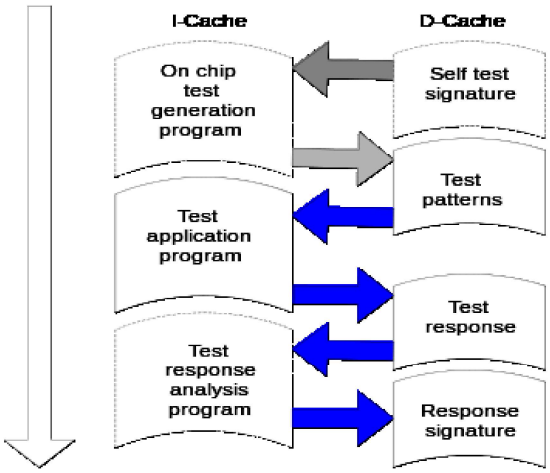


Figure 3.2.1: Software-based self-test of embedded processor cores. Self-test methodology

The test responses can be compressed into response signatures using a test-response analysis program. The responses are stored in the memory and can later be downloaded and analysed by an external tester. The method possesses the fault-coverage advantage of deterministic structural testing. The test application is performed at-speed if the component test application and the response collection are achieved with instructions instead of with scan chains.

In Bernardi **(Bernardi 2004)** work, this problem is exposed that hardware-based self-test procedures may negatively affect the performance of the processor. For these reasons software-based self-test methodologies seem to be better for testing embedded processor cores. Because the software-based self-tests consist of executing test programs, no extra hardware is required and no modification of the processors is needed. An economic benefit of the software-based self-test is in the reduced automatic test equipment control requirement and the better independence from the test frequency. A well-known problem is that each embedded core should be reachable from the top layer of the chip.

The basic idea of the self-test program is executed in an interrupt service procedure. Similarly my test program is dedicated to execution in an interrupt. The wrapper circuitry controls the interrupt signals by converting the high-level command coming from the ATE to the activation sequence of the processor-interrupt mechanism. The solution is presented in Figure 3.2.2.

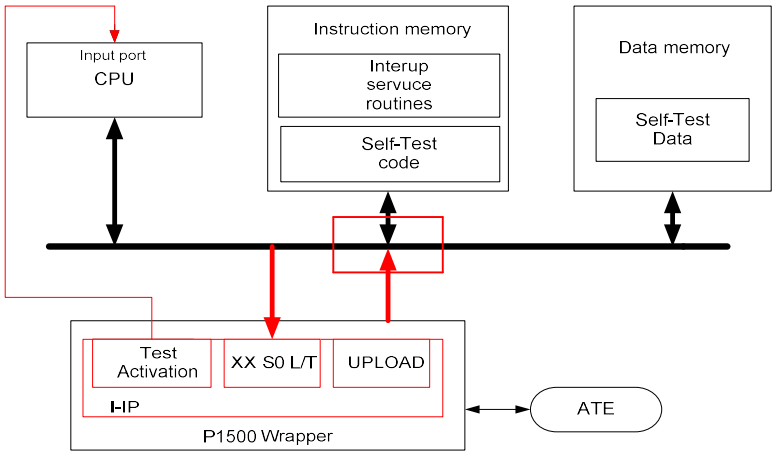


Figure 3.2.2: Software-based self-test of embedded processor cores. The architecture of the CPU

To overcome the problem of reach-ability of embedded cores many solutions use additional test-control components, the so-called wrappers. In reference **(Safi 2003)** an infrastructure IP (I-IP) is used to support a software-based self-test of the embedded processor cores. This solution is able to completely test the functionalities of the processor. Through I-IP the test program is uploaded to the instruction memory and the test procedures acting on the interrupt service procedure are activated. The structure of the approach is shown in Figure 3.2.3.

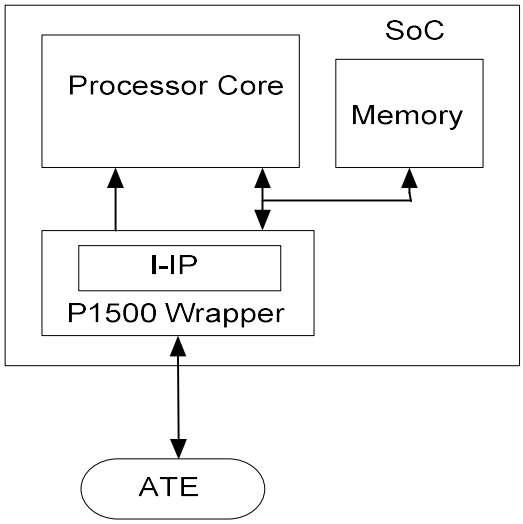


Figure 3.2.3: Software-based self-test of embedded processor cores. Test architecture with I-IP

The main features of the solution are as follows:

- no modifications to the processor's internal structure are allowed,
- the test must be performed at the same working frequency as the processor,
- a low-cost ATE is in charge of performing the test, through a low-speed interface,
- automated the generation of the final test program for the whole SoC starts from a knowledge of the test features supported by the composing cores is desirable.

A similar method of functional testing, proposed by E. Safi **(Safi 2003)**, is based on pre-computed structural test sets for the functional components of a processor. The Architecture Description Language (ADL) is used to collect information about the hardware. The information is used to implement a set of instructions, which is referred to as a macro. The difference of my approach **(Wegrzyn 2009)** consist in that the sub-sets of instructions are dedicated primary to test specified individual blocks of the processor. These macros are helpful at determining the input test vectors of the functional units and propagating the outputs of the functional units to the processor's output ports. The ADL language is used to describe the structure of a system composed of software and hardware components. It usually describes the functional interfaces of components for control and data flows and non-functional aspects of the components, such as timing and the safety level. The ADL language can be used at a higher level that describes how systems are composed

of components. The advantages of ADL are improved software, the system documentation, the flexibility and the modification of the design (Figure 3.2.4).

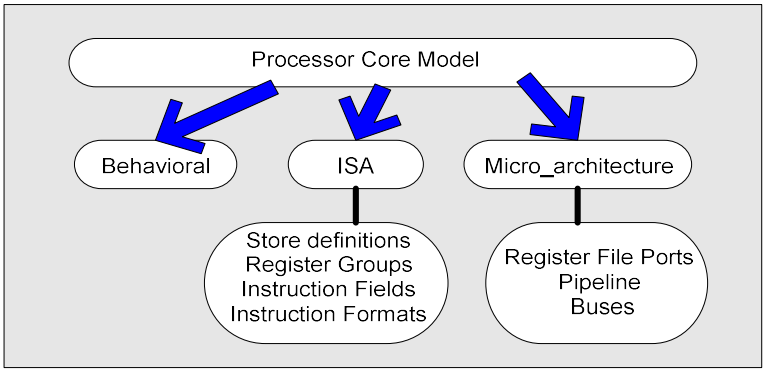


Figure 3.2.4: Software-based self-test of embedded processor cores. The features of an ADL model

In the ISA model each operation is described in terms of its opcode, operands and behavior. This is particularly interesting for the development of the method hereby presented. Each operand is classified either as a source or a destination. In the ISA model the opcodes and the addressing modes are included. They are important for defining the instruction formats. The fields provide information about the binary widths, the source versus the destination specification and the hardware components that the fields correspond to. The behavior of an instruction specifies its computational task, the assembly syntax of instruction, the mnemonics and the operands. The method relies on hierarchical testing using pre-computed deterministic tests generated by a commercial ATPG for the components in the data path. The set of instructions is implemented in such a way that it can justify pre-computed tests to the inputs of the functional units and make possible the observability of the results. The components for macro-generation can be classified into various classes as functional components (ALU, shifter, and multiplier) and storage components (registers).

The work of Mishra **(Mishra 2003), (Mishra 2004)** presents a test-generation method and a functional coverage estimation framework for pipelined processors using the so-called Specman Elite block presented in functional diagram in Figure 3.2.5 Verisity's Specman Elite is a comprehensive environment for all aspects of verification: automatic generation of functional tests, data and assertion checking, and functional coverage analysis. A random and constrained-random test program is generated in this block. The functional coverage estimation technique is developed in order to verify the quality of the generated test programs by the authors.

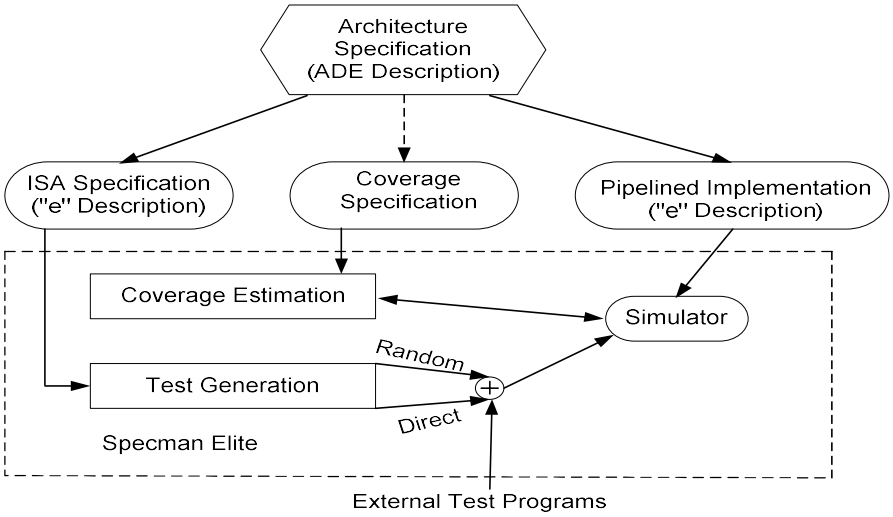


Figure 3.2.5: Software-based self-test of embedded processor cores. Test generation and coverage estimation

The solution generates ,,e" models for the pipelined implementation and uses an „e" simulator. The coverage estimation is used to measure the progress of the verification effort. Authors measured the functional coverage of pipelined processors. It evaluates the efficiency of the test programs used for simulation. The measurements cover:

- register read/write,
- operation execution,
- pipeline execution.

A slightly different problem concerning pipelining testing appeared during my research work, however studying of this paper was instructive.

The method presented by F. Corno **(Corno 2002), (Corno 2003), (Corno 2004)** exploits evolutionary techniques to automatically generate an effective test program. Some of the work **(Corno 2002)** is devoted to simulation-based design verification. For a specific RTL description of the microprocessor for a simulation-based verification they need a test program and a tool able to simulate its execution. The simplest method to implement a test program may seem to compile a high-level routine. Such a method relies on a compiler or a cross-compiler. According to the author this easy solution is usually inadequate to uncover any design errors due to the intrinsic nature of algorithms or the compiler's interpretation. A better solution is to implement assembler programs.

In the case of an assembler program it is easily to use incorrectly instructions. But an effort to implement a syntactically correct assembly source can be profitable. For the same reason my test methodology bases upon development of testing programs written on instruction level. In the case of the implementation of a random program, the main drawback of this method is that the program should be long enough to cover all corner cases. This results in a long simulation time. This method uses an evolutionary algorithm to construct a set of instructions. The basis of the method was as follows: the assembler program and the instruction set are constructed according to a certain order to achieve the best fault coverage.

Direct acyclic graphs can represent internally assembled programs. The evolutionary technique is very effective because it improves the achieved fault coverage. Moreover, the technique adapts the internal parameters to their optimal values. The proposed approach is illustrated in Figure 3.2.6 and in Figure 3.2.7.

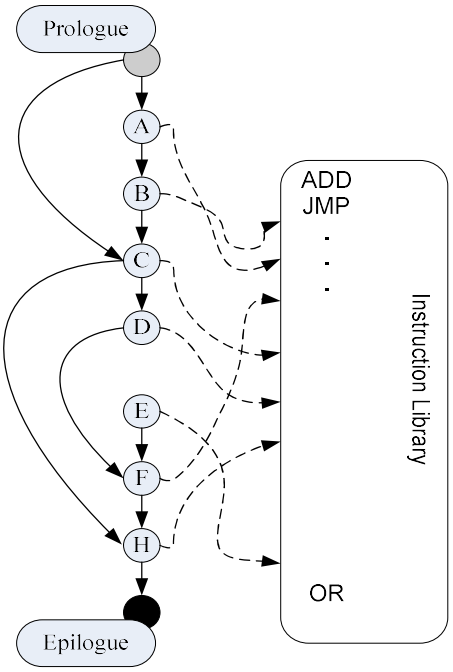


Figure 3.2.6: Software-based self-test of embedded processor cores: Directed acyclic graph and instruction library

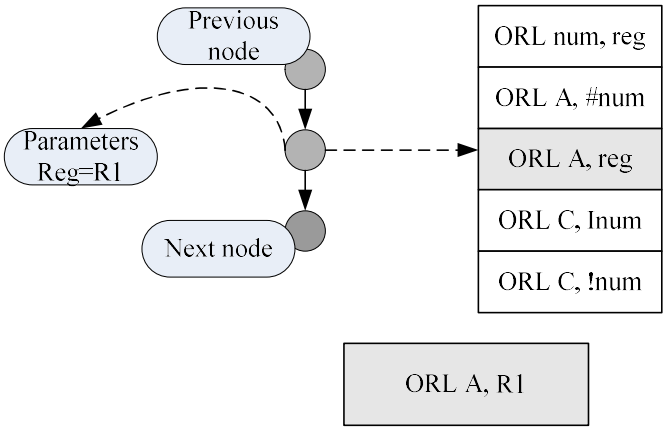


Figure 3.2.7: Software-based self-test of embedded processor cores**:** A sequential Node

The test program induced by the generator uses an external instruction library that describes the syntax of the microprocessor's assembly language. The generator uses a fault simulator in order to evaluate the generated test programs and in order to gather the necessary information for driving the optimization process.

The test-program generator exploits MicroGP, which utilizes a direct acyclic graph (DAG) to describe the syntactical flow of a program and an instruction library for describing the assembly characteristics.

The approach of **(Corno 2002)** repeatedly evaluates and improves the candidate test programs directly running on the microprocessor under test. The candidate programs are executed rather than simulated. This automatic methodology makes possible a rapid evaluation of the candidate test programs, even for large designs. An automatic program generator is run on the microprocessor under test, and then the candidate programs are executed and the feedback is analysed. So, the microprocessor under test has two tasks: it runs the test program and it evaluates the candidate tests.

The benefits of this solution are that the test-program-generation architecture can be implemented by a test engineer. As mentioned before, the functional testing of modern microprocessors requires knowledge of the architectural features and is a challenging task. Moreover, the test-program-generation process can be automated. An automatic test-program generator can be configured by a program builder, the description of the processor and an appropriate program evaluator. The developed verification programs, high-level coverage metrics can be applied as part

of a generation loop to ensure the suitability of the program. The solution for an automatic test-program generation consists of an automatic trial-and-error approach. The program-generation tools can utilize feedback in order to evaluate the candidate tests and improve them. The candidate test programs are executed on the tested processor, and the generator is run on a host computer. The main drawback of this solution is the limitation imposed by the feedback loop. Some modern microprocessors (e.g. Power PC, Intel Pentium, AMD Athlon, Sun UltraSPARC possess hardware-performance counters that allow monitoring events like cache misses, pipeline stalls, etc. Additionally, both the program generator and the candidate test programs can be executed on a tested microprocessor. The feedback information about each candidate program is collected in the microprocessor's performance counters **(Lindsay 2004)**. The benefit of this is that by extracting the performance counters' information it is possible to excite properly the specific units of a processor under test. The next advantage of the solution is the removal of the simulation overhead. This means that authors do not simulate a model of the tested processor.

There is a need for utilization an advanced testing environment during the main test phase. I considered the experiences of authors and focused my work towards development methodology, which makes possible composition a compact test program according of set rules. These rules are defined as a result of research work leading to efficiency evaluation of the test program in fault coverage meaning. My work is dedicated to both industrial and project verification testing, where such a test program can be easy and relatively quickly written by microprocessor designer, application engineer, and testing in case of critical missions, where occurrence of SEUs are probable. Then opportunities of connection with methods of diagnosing and partial reconfiguration are opening up. The PicoBlaze processor plays two roles: it generates reference results both on fault-free and with injected faults in its VHDL descriptions.

Bernardi and Sonza **(Bernardi 2014)** proposed an original hardware based technique for embedded microprocessor functional On-Line Self-Test. This concept is based on a Microprocessor Hardware Self-Test unit (MIHST), specially designed to this purpose. Its role is to generate and provide an instruction stream to the processor core. The MIHST also can observe the processor behavior.

The MIHST unit internally encodes the test program in an original way that exploits the test program regularity. This idea minimizes the hardware required to store test program.

According to the above ideas, the processor executes the instructions provided by the MIHST unit. Whereas the execution flow is not controlled any more by the processor. In the case of normal operation, when the processor executes the code of a conditional jump instruction, the processor

evaluates the condition and depending on the result it generates a different address during the following fetch cycle. It is different in the MIHTS mode; the following instruction is decided by the MIHST unit independently of the address generated by the processor.

An important advantages of the solution are:

- The test engineer can manipulate the address range, and improve coverage in this way.
- MIHST unit makes possible observation of the result of an instruction execution on the bus without having the rest of the test procedure compromised by any possible fault.

According MIHST functionality, the **execution flow** of the MIHST driven program is never altered by a fault occurrence. In a classical SBST schema some faults can cause an exception occurrence (e.g. erroneous access to memory). However, this problem disappears hereby, due to the fact, that the MIHST unit is forcing a predefined instruction stream that does not depend on the processor requests.

Hence, characteristic features of MIHST driven test are as follows:

- test always finishes,
- addresses are generated no longer by the processor, but by the MIHST. Processor receives instructions autonomously provided by the MIHST,
- the execution time is significantly shortened by removing the need for control instructions of the test flow, which is directly controlled by the MIHST unit.

Technical realization of the proposed approach is based on adoption of an Infrastructure IP (I-IP), called Microprocessor Hardware Self-Test (MIHST) unit, which is connected to the system bus. The interconnection of the I-IP within a processor-based SoC is supported by a multiplexer, as shown in Figure 3.2.8.

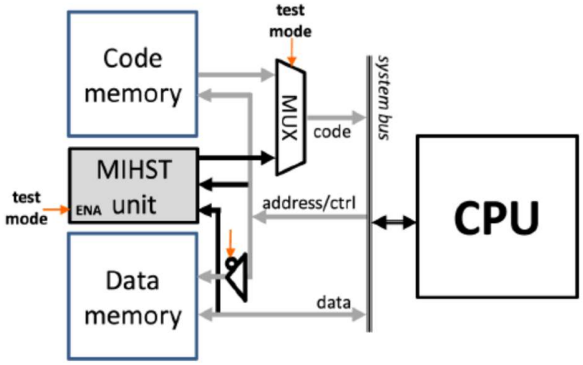


Figure 3.2.8: Architecture of system with built in MIHST unit

The connection of the MIHST unit does not require any modification in the processor's core, similarly to BIST, approaches **(Stroud 2004)**, **(Stroud 2003)**, **(Gizopoulos 2004), (Gizopoulos 1997)**.

This methodology is intended to application on-line testing in an interrupt in order to preserve a processor state. In normal mode, the code is executed in order of the addresses determined by the processor and thus the execution flow jumps back or forth according to the existing in code jump/branch instructions.

Riefert and Sonza **(Riefert 2016)** treat about a Software-based self-test (SBST) techniques. They described an automatic test pattern generation (ATPG) framework for the generation of functional test sequences, and an extension of this framework with the concept of a validity checker module (VCM). This (VCM) allows the specification of constraints with regard to the generated sequences. They applied the VCM to express typical constraints that exist when SBST is adopted for in-field test. The in-field test is a SBST test methodology, where restrictive constraints are imposed, e.g. the memory are available for the test program code and data may be limited **(Bernardi 2012)**, some input signals may hardly be controllable (e.g. reset), and only the final content of the memory can be observed. To face of the above, the generation of the applicable in-field test program is significantly more complex than for end-of-manufacturing test. Developed hereby solution **(Wegrzyn 2009)**, **(Wegrzyn 2014)** faces similar problems as limited data, especially in case of 32-bit microcontrollers, and phenomenon of excessively long time of exhaustive test.

Their study case **(Riefert 2016)** is a microprocessor without interlocked pipeline stages (MIPS)-like microprocessor. This method is able not only to generate a test program for mid-sized pipelined processors with high fault coverage, but this method introduces several optimizations which simplify the complexity of calculations and improve the fault coverage. This is an analogy to my solution.

I introduced bijective property for improvement of fault coverage **(Wegrzyn 2009)** and several optimizations at choice of test set in order to reduce memory resources, and shorten time of testing. A major advantage of this method **(Riefert 2016)** lies in the fact that it is also able to identify faults, which cannot be tested, when constraints are introduced.

A slightly different testing methodology, which I developed, identifies and lists faults which are untestable due to i.e. a hardware redundancy or detection of them is easy available, but can take excessive long time, and the importance for the functional test is insignificant. Other task of my method is evaluation of my test program. This program should detect almost all faults. If some test program cannot detect some number of detectable faults – this is its imperfection. On the other hand, if a fault does not manifest itself during normal processor operation – this is its positive feature. Such a list of untestable faults is helpful for analyzing of imperfections of a test program. This is dedicated primarily to evaluate developed by me testing program. In this work **(Riefert 2016)**, they shortened the runtime of the functional test generation algorithm by extracting and reusing knowledge collected during the test generation process and by implementing a heuristic for the reduction of aborts between testing. Developed by Riefert **(Riefert 2016)** methodology consists of steps of a fault sensitization: Structural Testability Check, Functional Testability Check, Test Sequence Generation, Partitioned Test Sequence Generation. Processing steps for fault sensitization are presented in Figure 3.2.9.

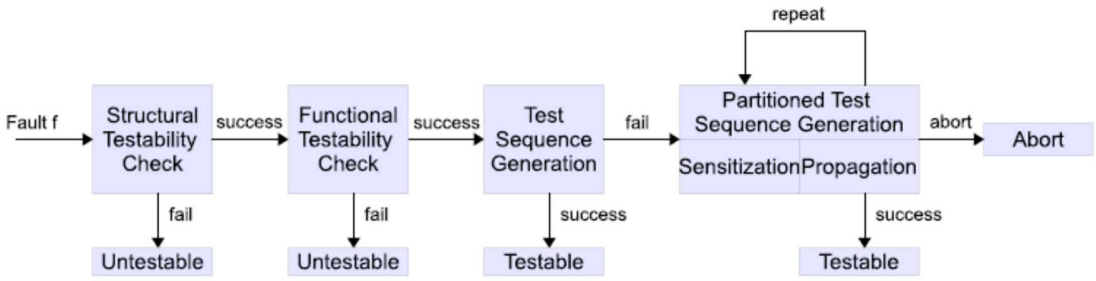


Figure 3.2.9: Processing steps for a fault sensitization

Testability of faults is checked and improved before execution of finally refined test, and two formal techniques as BMC (Bounded Model Checking) with Craig interpolation to this target are elaborated in **(Riefert 2016)**. In this paper, the solver Craig Interpolation Prover (CIP) described in reference **(Kupferschmid 2011)** is applied. Craig interpolants make possible exceeding approximate the reachable system states within each step.

The authors demonstrated, that Interpolation-based approaches can be very effective for targeting hard-to-detect faults and identifying untestable faults. Initially the ATPG framework starts with a fault list which contains all possible faults. Then all structurally equivalent faults are then removed. The removed faults from the fault list are processed one by one. If a Test Sequence (TS) is generated for a certain fault, all undetected before faults, which are detected by this TS, are removed from the fault list. This approach is analogous to the optimization methods which I developed.

The generation of a TS for a fault corresponds to elaborating an assembler code sequence, which tests this fault. Here certain similarities to my approach can be found. I looked for some code snipped to detect some particular faults, and I utilized outcome results as a new inputs.

Mentioned before heuristic is aimed at proper selection of flip-flops. The proper selection of appropriate flip-flops is very important for the success of these ATPG steps. For this purpose, a heuristic is pre-computed for each flip-flop, which estimates the probability of propagation of fault effect latched in the flip-flop to a primary output. This heuristic is computed by choosing several random functional states.

Evaluation of Proposed Optimizations:

In order to assess the effectiveness of the cache, Riefert **(Riefert 2016)** executed an additional evaluation run with no constraints applied and without utilizing the propagation sequence cache. Thus, the application of the propagation sequence cache improved the TS generation runtime by 28%. This demonstrates that the knowledge gained from already generated TSs can be automatically extracted and used beneficially for the further ATPG process.

Methodology developed by Tai-Hua **(Tai-Hua 2011)** is dedicated to software-based self-testing (SBST) of pipeline processor cores as miniMIPS and ARMv4. Instructions of processor are located in the main memory, thus the test process is accelerated and the observation of the execution results is available directly through the processor's local bus.

A fault simulation model is developed by the author to emulate the macro observation method. This model extends the mask circuit concept in such a way, that the hardware response signals affect correctness of the store operation. The fault coverage is measured on basis the results written in memory.

The developed test program consists of a deterministic test code, composed as a multiple-level abstraction-based methodology. Supplementary role plays here a random program based on a basic-block development method. I applied a supplementary random program in certain phase of my

experiments too. The deterministic test program explores the diverse design information of processor architecture, RTL, and gate-level for different types of processor components. Hence, the testing program applies the most useful information of a certain level to the specified parts of the processor core. Multiple-Level Abstraction-based (SBST) methodology is depicted in Figure 3.2.10 **(Chen 2007)**.

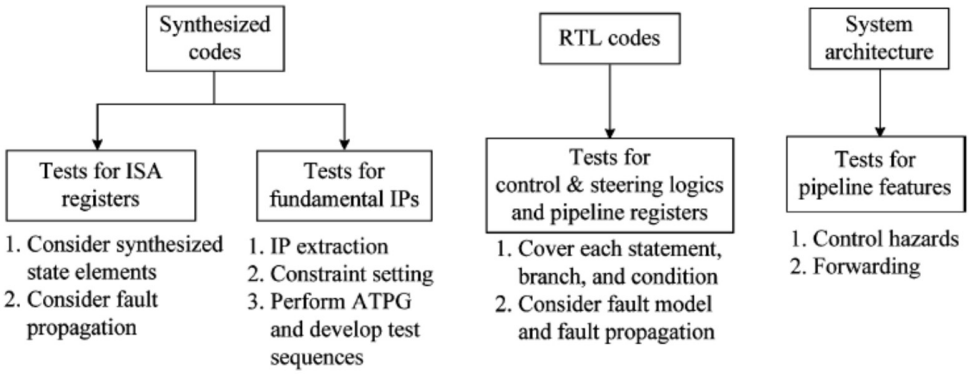


Figure 3.2.10: Multiple-Level Abstraction-based (SBST) methodology

Unlike to the construction of my blocks, each basic block contains every type of the instructions, except interrupt generating and exception instructions. Instructions for results observation are also included in every basic block. Processor modes are tested at usage of return block which triggers the testing on the access control of the shadow registers in various processor modes, including those for status registers. The circuitry for processor mode switching is activated by return block, interrupts and exception generating instructions. The switching mode is a critical function in modern operating systems. To support interrupt testing, here specific undefined instructions so called illegal instructions are applied in the processor ISA to trigger interrupt events through the test shell. The test shell is implemented usually as part of a common bus wrapper and thus introduces very small delay for memory access time.

To test processor interrupt mechanisms, author **(Tai-Hua 2011)** defined interrupt-request instructions through the undefined instructions of the processor (reserved op-codes). In this way, the processor ISA isn't changed. Usually when a random program is executed, the instruction fetch and data access addresses are unpredictable. This leads to fetch unknown instructions and access unknown data. Therefore, a simple hardware device called a test shell was elaborated and placed between the processor core and system bus as illustrated in Figure 3.2.11:

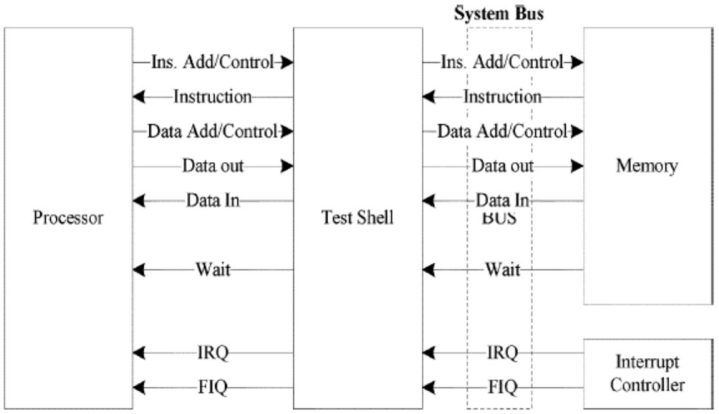


Figure 3.2.11: Test shell to sequential execution of random program

This device forces sequential execution of random programs. During my researches, an effort was focused on detailed analysis of interactions between hardware and test program instead of random testing and implementation of additional dedicated to random testing hardware. I achieved similar efficiency of fault coverage as Thai Hua. The choice of method may depend on a dedicated platform: FPGA, embedded or ASIC INTEL etc.

Authors **(Tai-Hua 2011)** used ModelSim as I did for logic simulation, Design Analyzer for synthesis, and Turbo Scan based on the stuck-at fault model for fault simulation in order to achieve experimental results. The model for research was ARMv4 ISA pipelined processor core implemented by authors and the miniMIPS processor.

4. Fault Injection

Fault Injection (FI) techniques are applied for effective evaluation and validation of test methods. Fault injection techniques are divided into simulation based and experimental. Both of them can be hardware-based or software-implemented **(Civera 2001)**, **(Sonza 2006)**.

Software-implemented techniques consist in fault injection by modifying the code executed by an application implemented on FPGA. I acted in this way in case of MicroBlaze. Other techniques consist in fault injection by modifying the HDL description of a circuit. I have designed and implemented such experiment in case of PicoBlaze. These techniques allow evaluating complex systems. Some of the solutions use debugging exception mechanisms to trigger and inject faults, others use for example the interrupts of microprocessors or Code Emulated Upset (CEU).

Hardware-based techniques utilize a VHDL models implemented in FPGA architecture. These most often used approaches are based on inserting faults in the high-level VHDL model of the circuit and then analysing the results produced by the faulty model. For example work **(Alderighi 2003 A**) presents a fault injection approach for SRAM-based FPGAs. The fault injection is performed by modifying the configuration bit stream while this is loaded into the FPGA's configuration memory, without using tools as Jbits. The configuration bit stream is modified while it is being loaded into the DUT (Device Under Test). The fault injection relies on flipping the logical value of the configuration bit trough a byte wide „XOR" logic gate. It is controlled by a Fault Injection Register.

The approach **(Alderighi 2003 A)** doesn't use a standard synthesis tools, commercial software etc. I am taking a step forward, I inject faults in structural level of VHDL processor description **(Wegrzyn 2009)**. When available is high-level VHDL model, it is easy to translate it into structural level by usage of Xilinx tools. Whereas I avoid not recommended by among all M. Sonza Reorda modification of the configuration bit stream.

The solution **(Alderighi 2003 A)** is different from my method because the latter utilizes FPGA hardware; however experience, which has been gathered in this work, may be helpful for further development for other devices.

Moreover the solution **(Alderighi 2003 A)** focuses on the bit-flop fault model what corresponds the modification of the content of a memory element during the execution. The faults are located in the programmable logic and embedded microprocessor.

Other solutions **(Alderighi 2003 B)** of Fault Injection applied to Xilinx SRAM-based FPGAs, doesn't utilize commercial tools as Jbit too. The faults can be inserted directly into the configuration bit stream downloaded into the device. Precisely, faults are injected only into configuration of memory cells and user registers. These solutions allow a more realistic study of device behaviors and take less time. That's the advantage of them. Usage of Jbits is limited because not every FPGA resource can be directly altered in this way for example Programmable Interconnection Points. Moreover if we haven't the Jbit tool, we should not try to modify a bit stream without knowledge about which slice of bits is responsible for. Otherwise we can damage an FPGA chip.

Hardware based techniques speed up significantly the fault injection process and they may exploit modern FPGA architecture to emulate system composed of thousand gates for example in the case of embedded processor cores. The drawbacks of the technique are: very long time of simulation in case of complex architectures and the hardware overhead in order to reach the targets of fault injection.

Simulation-based techniques consist in simulating the Device Under Test (DUT) and injecting faults in a simulated model. The faults are injected by modifying the logical values during the simulation. A specific simulators to inject faults are known from bibliography. Many others methods consist in simulating a model described in a HDL language by using commercial simulators too. A possible solution for my study case is PicoBlaze **(Wegrzyn 2009)**, **(Wegrzyn 2014 A)**. Simulation-based techniques allow early and detailed analysis of designed system and can be applied when a prototype is not available. Often instead prototype, merely IP core with functionality of the prototype is available. These techniques make possible early and detailed analysis of practically any possible fault in a designed system.

Experimental techniques are useful, when a prototype of the system is available. In situation, when a prototype is not available i.e. when Intellectual Property (IP) is protected, simulation based techniques are utilized. Intermediate solutions are often used in practice, because they collect benefits of experimental and simulation based solution. The FPGA technology can be exploited to perform Fault Injection experiments. The Fault Injection process is performed during reprogramming

of FPGA.FPGA-based Fault Injection techniques are often more convenient than until now known classical hardware-based, because they may allow the injection of a wider set of faults. The technique opens up possibility of injection before specification of faults inside the circuit. There exist solutions, which do not require FPGA reconfiguration for each fault experiment, so they are significantly time-efficient **(Civera 2001)**, **(Hayes 2007)**. Such solutions perform additionally good observability.

Practical FI solutions **(Leveugle 2004)** are based on the injection of faults in behavioral descriptions of blocks. This high level description is usually done in VHDL or Verilog languages. The fault injection can be executed by means of simulation or emulation and the resulting traces are used for classification of the faults with to pay respect to their impact on the behavior. All information required for the fault injection is provided during the campaign of definition by delivery. Two different approaches to a problem of modification the initial description of the circuit are known from bibliography. The first way consists in insertion between the existing blocks, an additional block which can corrupt signals in order to emulate some kinds of faults. These signal corruption blocks are called saboteurs in bibliography. This method is easy to implement and requires only modifications of some interconnections in the initial description. Unfortunately, saboteurs can inject faults exclusively on these interconnections and it is almost impossible to inject higher-level (behavioral) errors or to modify signals within the initial blocks that is drawback of this method. Second way consists in direct modification of a signal inside the block and block function. It is more approximate to and more powerful, but it is much more difficult to implement. Its modified description is called mutant **(Leveugle 2004)**.

According to similar principles as above, circuit is transformed usually to easily modify the value in elements of the circuit memory. The method presented in **(Civera 2001)** is convenient to trigger the occurrence of faults at the time of injection and to observe the faulty behaviour of the circuit. The circuit of more developed architecture is equipped with a Mask Chain register, added to the original circuit (see Figure 4.3.1). It stores the binary information about, which flip-flops should be affected by the fault, and which additional logic should be used to perform the Fault Injection. The signal, inject controls the Fault Injection. The value of this signal is set by the Fault Injection Master to force the selected bit-flops.

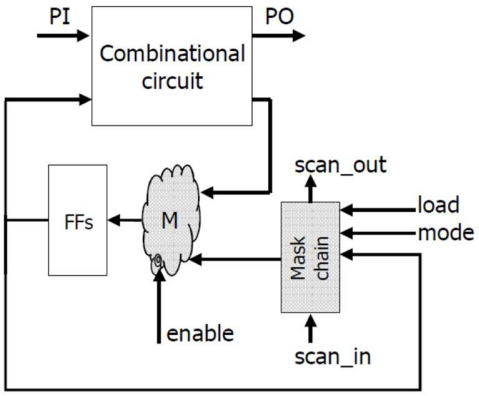


Figure 4.3.1: Instrumented circuit

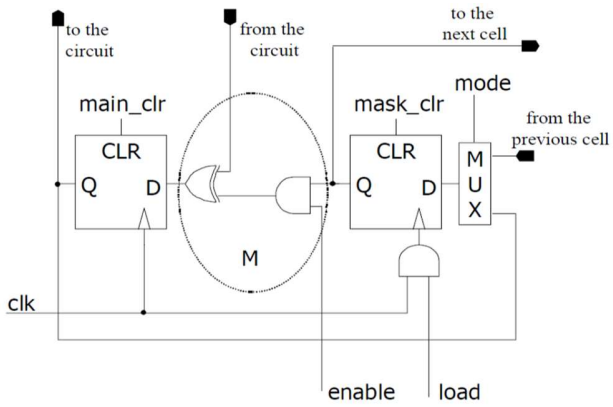


Figure 4.3.2: Instrumented flip flop

There are following additional modules added to the circuit:

- Mask Chain: the value of each bit in FFs is determined by a Mask Chain, which is a parallel - and serial-load register. The signals load and mode control its operations.

- M is the combinational logical part utilized to complement the output of the Combinational Circuitry that is loaded into the FFs module. The contents of the Mask Chain module and the inject signal determines the behavior of M.

Detailed description of this in what manner each flip flop in the original circuit is modified, is presented in Figure 4.3.2.

There are implemented two systems with and without faults on a FPGA board. The FPGA board is driven by a host computer. The proposed in **(Civera 2001)** approach utilizes a FPGA board that emulates the gate-level system with and without faults. The FPGA board is driven by a host computer. Other modules composing the Fault Injection system are also commanded by the host computer. The Fault Injection environment is located on the host computer. Fault Injection environment consists usually of such modules as:

- Fault List Manager which analyses the system and generates the list of faults to be injected.
- Fault Injection Manager which manages the selection of a new fault, its injection in the system and results observation.

- Result Analyzer, which analyses the data obtained from the experiment, defines categories of faults according to their effect and produces statistical information.

In order to determine the behaviour of the circuit when a fault appears, the FPGA board emulates an instrumented version of the circuit, which allows both the injection of each fault and the observation of the corresponding faulty behaviour. The fault injection environment is described more in details in Figure 4.3.3.

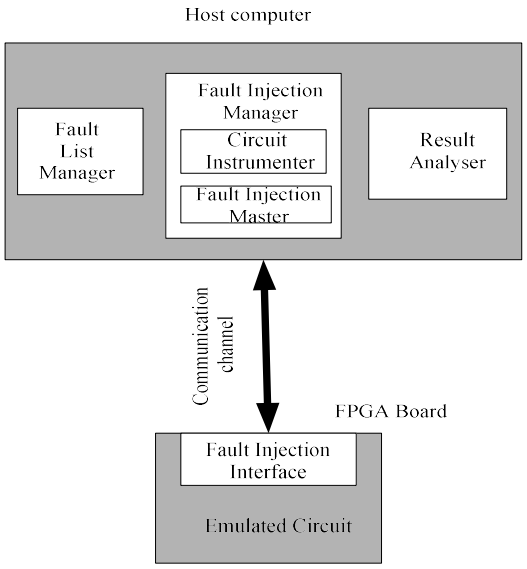


Figure 4.3.3: Fault Injection environment architecture

The fault injection manager consists of the following parts:

- Circuit instrumenter; generates certain version of the circuit description, which is downloaded on the FPGA board for performing Fault Injection. The circut instrumenter is realized by the author **(Wegrzyn 2009)**, a little different as in **(Civera 2001)**, by preparation of PicoBlazeVHDL code by the perl script for simulator.

Fault Injection Master has more tasks:

- Downloading to the FPGA board the instrumented description of circuit **(Civera 2001)**. This is realized in **(Wegrzyn 2009)** by loading the fault free version of PicoBlaze VHDL code by the perl script to simulator.

- setting up the environment for the Fault Injection process **(Civera 2001**). At **(Wegrzyn 2009)** a CADENCE NC VHDL tool commanded by a perl script is setup.
- repeated access to the fault list, selecting a fault list, selecting a fault and sending to the FPGA board of the information for its injection in **(Civera 2001)**. At **(Wegrzyn 2009)** Perl script injects faults by changing a parameter in VHDL description. Faults are selected one by one.
- launching of circuit emulation, by providing the input stimuli and the triggering signal for the injection of a fault, retrieving from the board the information about the behaviour of the faulty system in **(Civera 2001)**.

At **(Wegrzyn 2009)** this is controlled by the pearl script. The input stimuli are provided together with test program, and placed in dedicated PicoBlaze ROM memory. Simulator generates output result file. Methodology developed by the author **(Wegrzyn 2009)** can be partitioned into the same three steps as is proposed by **(Civera 2001)**. However the main difference is that my method is exclusively simulation based in case of PicoBlaze. The simulation based method was faster than the solution where PicoBlaze was implemented in FPGA board and Fault Injection Manager placed on host PC. Research equipment, I had in Jozef Stefan Research Institute at this time, limited significantly speed of communication between FPGA board and PC. Initially I led such experiment with a processor implemented in FPGA, but it was very time-consuming due to reprogramming FPGA board, every time with different, modified by a fault configuration, and communication speed. My intention was to develop exhaustive test, where complete set of test vectors was transferred to PC. Then all outcomes were transferred back to the PC and elaborated there. Time of sending data through serial port, and receiving results through parallel port was significant too. I have written a C program for Windows to handle the parallel PC port for this purpose. Then for another experiment, I prepared a file with list of faults. My Fault Injected Manager is implemented as a pearl script and Result Analyser consist of script programs written in C according to principles described in chapter 7.

The System on Programmable Chip (SoPCs) include processors, memories and programmable logic that allow catching multiple application requirements such as high performance and reconfigurability. The approach described in the paper **(Alderighi 2003 B)** exploits the programmable logic in the SoPCs to implement a fault-injection module, so called Fault Injection Hardware Unit (FIHU). The unit makes possible fault injection in the different components of a SoPC and the observation of their effects. Whereas in case of my solution, HDL description of microcontroller core is placed inside only one

file, and every its block is described in the same way. Thus, to provide fault injection to any component of microcontroller core doesn't constitute a problem. The pearl script does not need to distinguish between block of the core.

Authors **(Alderighi 2003 B)** chose the Select Map configuration mode of Virtex FPGA because the Select MAP interface can be easily driven by a processor or another FPGA that manages the configuration and read back dedicated pins.

There is a built-in finite state machine which handles device configuration. The cyclic redundancy code (CRC) value is checked against an internally calculated CRC value. **When the CRC error has occurred the FPGA becomes inactive.** In the opposite case the FPGA becomes active and operates with the loaded design. A functional diagram of the FPGA control mechanism is depictured in Figure 4.3.4.

The internal configuration memory is divided into a number of frames. So, the portion of the bit stream which is loaded into configuration memory consists of data frames. The remaining part of the bit stream contains information that drives the correct operation of the built-in finite state machine (FSM). The number and size of frames depends on device size. After that, when internal configuration registers were accessed and loaded by the common configuration bus, each phase in this flow is accomplished. Writing a register occurs in general in two steps:

- first is a 32 bit command header that constitutes a sort of instruction to be interpreted by the built-in FSM,
- second is the actual datum to be loaded into the addressed register.

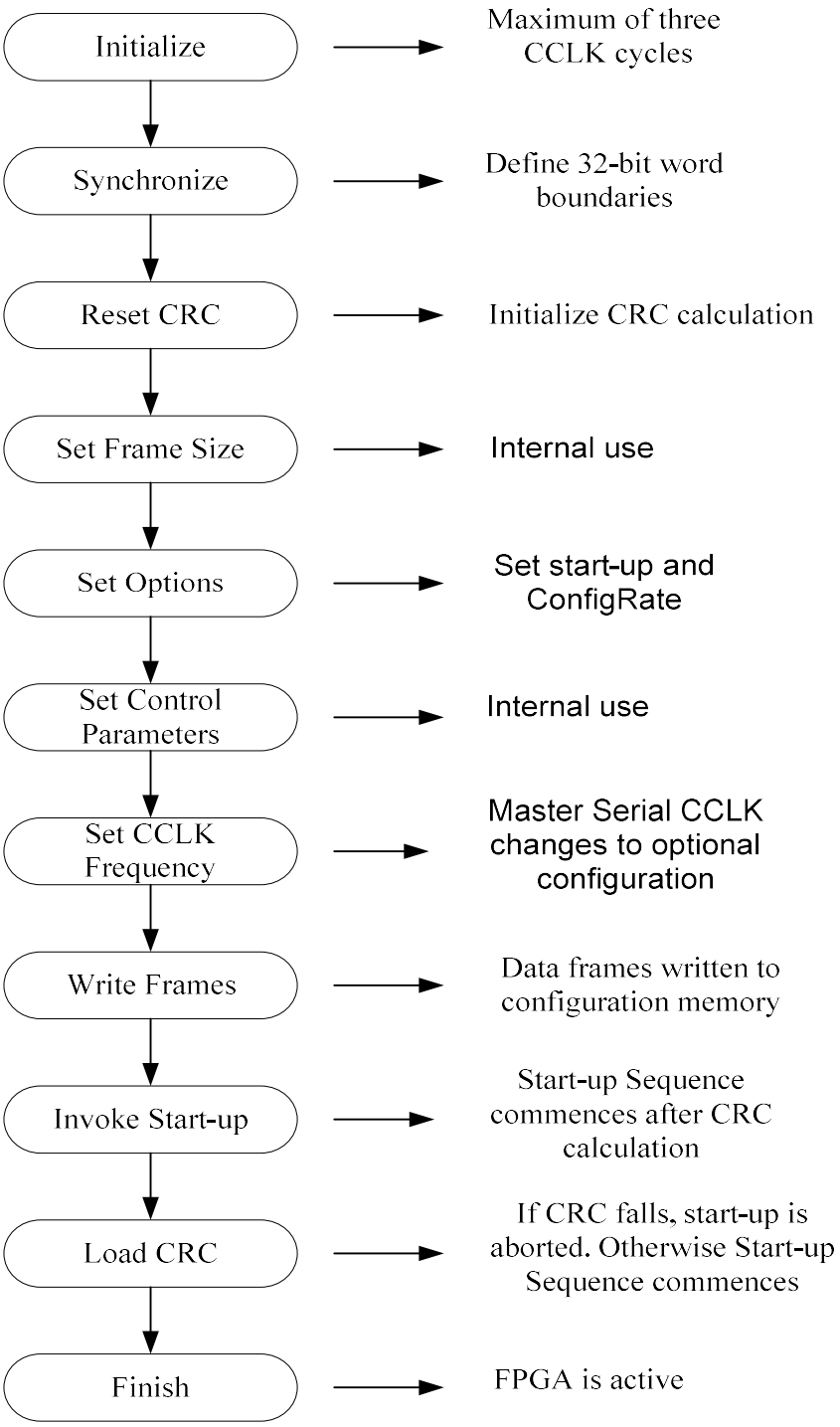


Figure 4.3.4: Configuration control mechanism

The Fault Injection System overview is depictured in Figure 4.3.5. And the realization of the Fault Injection Tool is presented on Figure 4.3.6.

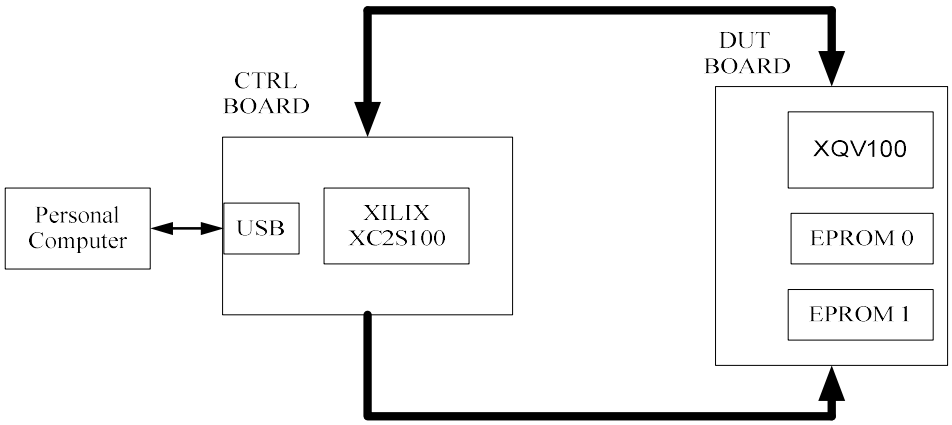


Figure 4.3.5: System overview

The operation of the whole fault injection system is controlled by the configuration control logic (CONF CL). At first the non-faulty bit stream is loaded into the Configuration Memory. The bit-stream is modified when loading into the DUT. For modification of the bit-stream one byte wide XOR is utilized. It is realized by simple flipping of the logical value. The operation is controlled by a Fault Injection Register (FIREG). The CONF CL increments the ADD GENERATOR counter, which addresses both configuration memory and mask memory. I had lack of knowledge about configuration bit-stream structure. However I utilize similar technique in case of MicroBlaze assembler program, instead of modification of the configuration bit-stream. I use XOR mask between assembler instructions. The technique proposed by myself is safe, because cannot damage FPGA in any way, and can be applied alternatively instead of the modification of bit-stream. This technique is described in chapters 5.2 and 6.3.1.

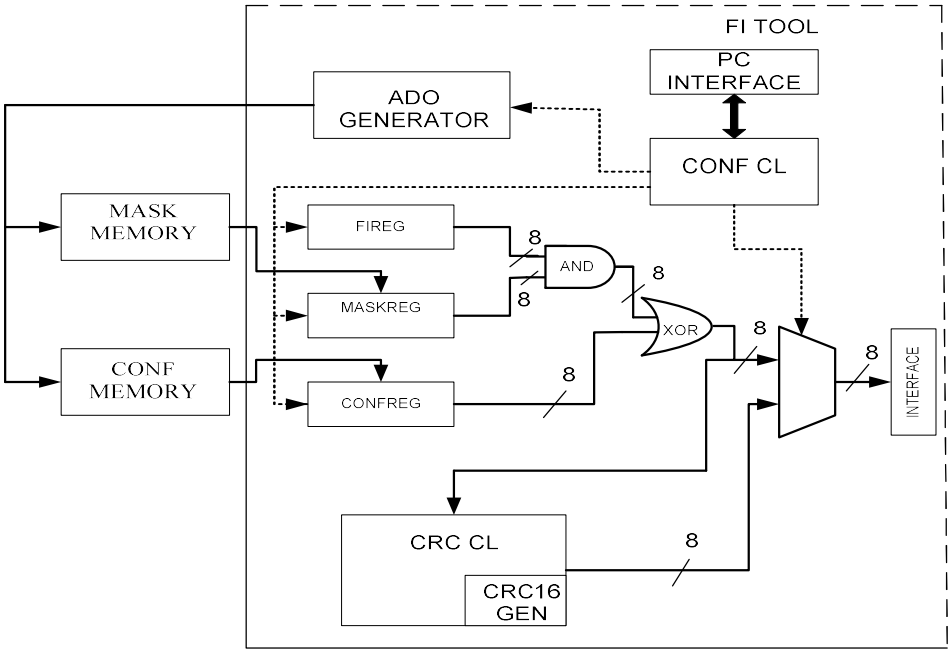


Figure 4.3.6: Fault Injection tool

In order to make it possible the faulty bit stream to be loaded and to prevent FPGA from signalling a CRC error, the actual value of the CRC is recalculate. The new value CRC is computed by the CRC logic block (CRC CL) and actual data for CRC generator are arranged.

The approach focus on SEU-like faults affecting the configuration control mechanism of FPGA and permanent stuck-at faults.

5. Proposed Solution: Sensitive-Path Approach

5.1. Basic principle

The developed experiments are targeted at maximal fault coverage, achieved by the developed test program at its as compact as possible architecture. The idea **(Wegrzyn 2014 A)** is to use appropriate microprocessor simulator which accepts its specification in HDL language, correlates it with the targeted FPGA, performs simulations with provided programs (in assembler) and allows analyzing the behavior of the tested application (e.g. program results) in this environment. These assumptions were performed by two simulators: Cadence NC VHDL and Mentor Graphics ModelSim. Fault injection is performed at microprocessor HDL structural description level, which reflects FPGA implementation.

In the proposed approach, the goal is to generate a compact test sequence that detects permanent SEU-induced faults of embedded processor cores in SRAM-based FPGAs **(Wegrzyn 2009)**. As described in **(Safi 2003)**, the functional model of such faults differs considerably from the conventional stuck-at fault model due to the fact that SEU-induced faults affect logic elements implemented by the look-up tables, in this way that the logic function is arbitrarily changed. While the existing fault simulators do not cover such a functional fault model, I follow the implicit strategy of test adequacy and statistical testing **(Zhou 2006), (Sosnowski 2005)**. I generate a test sequence that allows arbitrary situations that might occur in practice and consequently detects faults that only appear in a particular sequence of events. This is accomplished by using a test sequence that explores the functionality of each individual instruction and is composed in such a way that it forms a sensitive path. This path can be executed more than once, each time with a different input pattern **(Wegrzyn 2007)**, (**Wegrzyn 2009)**.

Although I have borrow the notion of a sensitive path from the automatic test-pattern-generation (ATPG) techniques **(Doumar 1999)**, **(Renovell 2001)** in my case it has a slightly different meaning

(Wegrzyn 2009). The path sensitization in conventional ATPG techniques for automatic test generation involves the generation of the path that is sensitive to the presence of a stuck-at fault and the justification of the values along the path by propagating signals back to the primary inputs. The key achievement of this work is proposed bijective testing procedure (further described in chapter 5.3) for which the fault detection is performed at the instruction level by a compact test program in which individual processor instructions are organized in a such sequence that the destination register operand of *i*-th instruction represents the source register operand of (*i*+1)-th instruction. In the test sequence, each processor instruction participates at least once (in order e.g. to test the instruction decoder of the processor core). The principle of instruction sequencing is presented in Figure 5.1. Intuitively I assume that the test sequence represents a sensitive path if the data flow through it is sensitive to changes of the input pattern. I pursue the following two goals:

- The faults occurring during the execution of individual instructions in the test sequence should manifest themselves in the final result,
- In order to increase fault coverage, the data-sensitive path should provide a way of randomizing the instruction operands of the test sequence, resulting in increased processor activity and consequently in increased fault coverage.

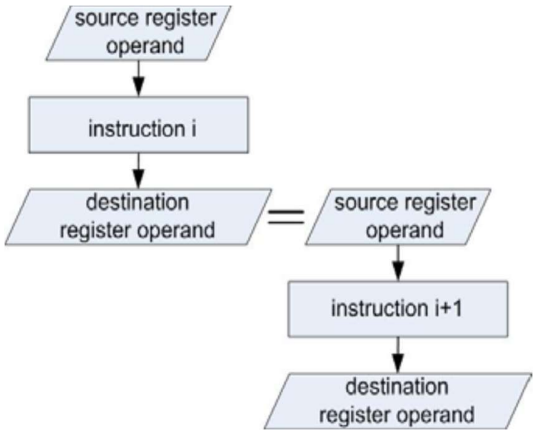


Figure 5.1: The principle of instruction sequencing **(Wegrzyn 2007)**

A data-sensitive path can be achieved by the following two basic principles of design-for testability: controllability and observability. Controllability is the ability to set the values of the inputs of any system component from the primary inputs of the system. Observability is the ability to observe

the values of the outputs of any system component at the primary outputs of the system. An instruction of a test sequence can be regarded as a system component. The test sequence is composed of individual instructions (i.e. system components), which act upon the data stored in registers and memory cells. **An instruction processes the input data (i.e. the argument) and generates a result that represents the input data for the next instruction in the test sequence.** The input data of the first instruction of the test sequence represents the system's primary inputs, while the results of the test sequence system are the primary outputs. The test sequence is composed in an incremental way: each time a new instruction is added to the test sequence and the resulting test block is checked for controllability and observability **(Wegrzyn 2009)**.

The requirement that the test sequence preserves a sensitive data path between the input data and the result is a prerequisite for achieving high fault coverage. On the other hand, some faults may still escape if the input data does not lead to the occurrence of an event that would manifest itself in a result that is different from the expected reference obtained on a fault-free system.

In order to detect these faults I can re-run the test sequence with different primary input data, **(Wegrzyn 2007)**.

5.2. Initial approaches

Initially for my case study, the MicroBlaze microprocessor was chosen, and the approach was implemented for a functional test of this popular soft-processor core supported in the Xilinx series of FPGAs. First experiments were led using a Xilinx FPGA board with downloaded MicroBlaze bit-stream. The assembler test program and a loop to input patterns generation were implemented on the host computer as described detailed in chapter 7.1. The initial idea was to create of a data sensitive path by invention of such an assembler program, which preserves all data. In other words, neither input data on any bit nor status flags are lost. It was realized by bonding all sub-blocks and program instructions by „XOR" instruction. As an important measure of the test program quality, I proposed the principle according to which the program should generate an unique result for each input test vector. In such a case, I assume that all sensitivity paths should be activated and any faults should not be masked. However, both the designing and evaluation of a test program meeting the above principle proved to be difficult in practice. Generation, transmitting to FPGA, and receiving all 2^{32} of 32-bit vectors turned out to be extremely time consuming due to communication between FPGA

board and the host computer. Physical induction of errors resulting from exposure of the FPGA system to radiation turned out to be impossible due to the lack of appropriate equipment at the department of the research institute Jozef Stefan in Ljubljana, Slovenia. Simulation based fault injection turned out hard to be realized too, because the MicroBlaze is available as IP core, and HDL description is not available. These problems are described with details in chapter 7.5 a describing MicroBlaze. Difficulties described here disappeared when I took PicoBlaze microcontroller instead MicroBlaze. That's the reason I developed end evaluated the test program for PicoBlaze first. The test program for MicroBlaze can be easily implemented according to the proven rules, which worked well in the case of PicoBlaze. Some MicroBlaze test program examples are presented in chapter 7.2, 7.5 and 7.6, (about MicroBlaze). Development of PicoBlaze test program is detailed described in chapters 5.3, 5.4, 6.3.

5.3. Design, researches and evolutions of PicoBlaze test program

The first program for testing PicoBlaze was developed applying rules described in chapter 5.2. Hence, all program instructions and sub-blocks were bonded by „XOR" instruction. The bijective property should be ensured. Assurance of bijective property of the whole program composed in this way turned out to be a difficult task. There were too many overlapping sub-problems, related to all the status flags, results with fixed values reminded still on the some bits, branch instructions execution etc. Thus, such the solution relying on application merely „XOR" instructions as data connectors turned out not very effective, and gave a reason to further investigations. Results of detect-ability of this program were compared to bibliography **(Corno 2002)**, **(Bernardi 2004)**. Moreover a few additional experiments were led in order to compare the number of detected faults with a specific applications programs, i.e. with my programs which generate Fibonacci sequence **(Wegrzyn 2009)**, matrix multiplication applications **(Wegrzyn 2014 A)**, etc.

The Fibonacci sequence is the series of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,…, where the next number is calculated by adding up the two preceding numbers. I wrote two different programs to calculate Fibonacci sequence in PicoBlaze assembler. The first one using an iterative method, and the second one using recursion.

These matrix multiplication applications check FPGA fault susceptibility in practical case, when the processor resources can be used partially or in a limited way. Thanks to these applications I can encounter natural fault masking capability of the application as well as I introduce some additional fault tolerance mechanisms at the software level. For this purpose I have developed three matrix multiplication programs (MM1-MM3). The basic program MM1 comprises 133 instructions, only 16 different instructions from the PicoBlaze instruction set are used. Program MM2 is an enhanced version of MM1 by adding control sums mechanism. MM2 program comprises 226 assembler instructions, using 17 different assembler instructions from the processor list. Program MM3 is a version of MM2 supplemented by exception handling. It comprises 386 instructions, using 21 different assembler instructions from the processor list.

5.3.1. The idea of facilitating: the compact test program composed of bijective blocks

The controllability and observability principle, proposed in earlier publications utilized in industrial testing, is only a vague concept that leads to different implementations of the test sequence with a relatively diverse fault coverage. Instead of introducing some kind of metrics as a guideline to efficient solutions, I impose a stricter rule on the test-sequence generation by requiring that there is a one-to-one, i.e. bijective, correspondence between the input test pattern and the result.

5.3.2. Bijective function

Definition: In mathematics **(Wikipedia 2020 A)**, a **bijection**, **bijective function**, or **one-to-one correspondence** is a function between the elements of two sets: X, Y, where each element of set X is paired with exactly one element of set Y, and each element of set Y is paired with exactly one element of set X. There are no unpaired elements. In mathematical terms, a bijective function $f: X \rightarrow Y$ is a one-to-one and onto (surjective) mapping of a set X to a set Y. Figure 5.3.2 presents bijective function.

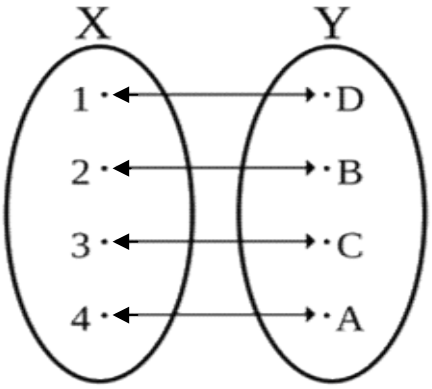


Figure 5.3.2: Bijection

If I apply this rule at the level of sub-sequences of the instruction sequence I can ensure high controllability and observability within the whole instruction sequence, which is a prerequisite for achieving high fault coverage.

5.3.3. Refinements to achieve full bijectivity

For some instructions the output data may not be completely sensitive to every change of input data and hence the property of a sensitive data path is not preserved. For example, some part of the register holding the result of the instruction operation may be cleared or set to all 1's. In such a case additional data manipulations need to be performed (i.e. the input data is stored at another location and logically combined with the result of the executed instruction). Summarizing, bijectivity is closely related to the full flow of information through the test program.

The flow of information can be disturbed by:

- incompetent composition of a test program, which does not provide full flow of information,
- masking the flow of information related to problems that are not completely solved due to the overlapping of flags generated by different instructions, operation of different instructions on the same registers and data – to be solved by a programmer,
- nature of SHIFT instructions, (by execution merely „SHIFTs" instructions, not full range of numbers is generated. Unless we use special solutions as LFSR),

- masking the flow of information related to processor implemented in FPGA hardware construction as delays **(Abramovici 2002)**, hardware redundancies **(Renovell 2000 B)**, simplifying the construction of individual sub-blocks of the processor.

For illustration, a part of the test sequence organized in a data-sensitive path is shown in Figure 5.3.3. The destination register operand of the instruction represents the source register operand of the next instruction in the test sequence.

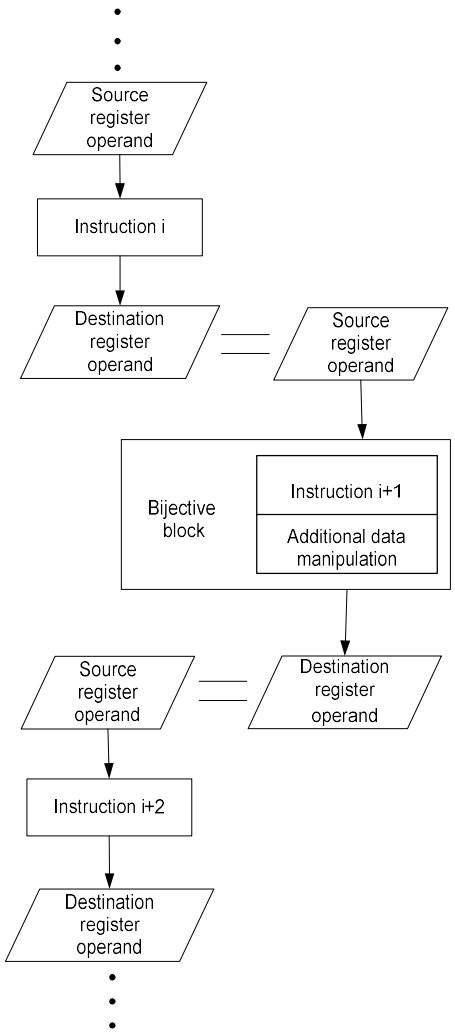


Figure 5.3.3: My Solution-Sensitive-Path Approach. A part of the test sequence

The execution of some instructions affects the status flags (like, for example, the ZERO flag, CARRY, etc.). In order to detect possible faults in the status information, the contents of the status register are included in the result of the currently executed instruction. This is usually achieved

by „XOR-ing" the contents of the status register and the resulting output data. However, more complex operations in assembler are applied as described further in this chapter, in the case when alone „XOR-ing" does not work. With such refinements the instruction and additional data manipulation code represent a bijective block within the test sequence. The basic architecture of a bijective block is presented in Figure 5.3.4. Bijective property opens up possibilities of further optimizations such as cyclic usage of output results as indicated by the dashed line in Figure 5.3.4.

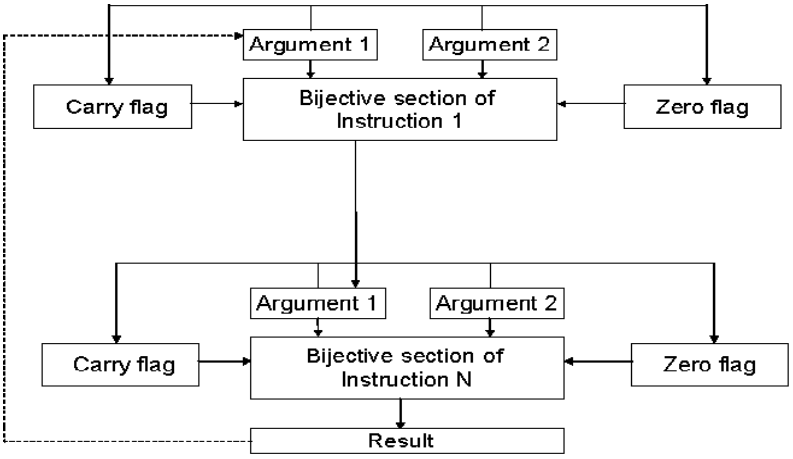


Figure 5.3.4: Architecture of bijective block

Test sequence is composed of bijective blocks. **By definition any program composed from bijective blocks is bijective**. A bijective block can be a single instruction if it exhibits bijective property. If not, some additional data manipulation is required to obtain a bijective block. I have found several ways to achieve bijective property of the PicoBlaze assembler instructions. These ways are:

1. **IDENTITY**,
2. **Continuous ADDITION or SUBTRACTION of constant value e.g. „1",**
3. **Flag register** (e.g. carry flags) **generation or recovery** (on basis of actual data),
4. **Negation** (e.g. by „XOR-ing" data),
5. **Bits permuting** (e.g. ROTATE data),
6. **Look Up Table (LUT) method** (not used hereby),
7. **LFSR**.

An **identity** function in mathematics is also called an identity relation, identity map or identity transformation. This is a function that always returns the same value that was used as its argument. That is, for f being identity the equality $f(x) = x$ holds for all x.

Several instruction as INPUT, OUTPUT, STORE, FETCH , TEST, COMPARE, satisfy the identity function by themselves without interference from the programmer. Additionally, it is possible to transform such an instructions as: AND, OR to satisfy identity relation.

Differently, in case of such instructions as ADD, SUB, I can apply such a simply method by **adding / subtracting „1''** (or in general case a constant value) to register (ADD / SUB). Additionally for instructions as ADDCY (ADD with CARRY), SUBCY (SUB with CARRY), I need to **generate input CARRY** by preceding instruction. Such SHIFT instructions as SLA, SRA (shift left/right arithmetic) capture input CARRY too, hence its previous generation is necessary.

ARITHMETICAL instructions as: ADD, ADDCY, SUB, SUBCY and all the SHIFT instructions set CARRY flag, and the need to recover of information contained in it appears. The XOR logic instruction constitutes mutually unambiguous transformation according to mathematical classification. This means multi-valued from its definition (bijective). In informatics XOR is often used for bitwise operation.

The ROTATE instruction constitutes mutually unambiguous transformation. For every different input data, I have obtain different unique outcome. Operation of rotation does not lose data. All the details, how PicoBlaze RR (Rotation Right) and RL (Rotation Left) operate, are described in chapter 5.4 and PicoBlaze User Manual (attached CD).

Permutation method is applied to testing some of „SHIFT" instructions. The method consist in rotations all bits, rotations of „CARRY" flags, I do not change the functionality of any instructions at the hardware level in reality. Such instructions as SHIFTs with „0" or „1" fill constantly on MSB or LSB position cannot be bijective alone. This problem can be solved by using additional instructions, which ensure continuous data flow through all bits of a given register. In this way an individual SHIFT together with these auxiliary instructions create a bijective block of instructions. The permutation method is applied in testing SR1, SR0, SL1, SL0, SRX, SLX (Shift register right/left, one fill, zero fill or sign extended) instructions, and detailed implementation of every block is described in chapter 5.4.

The **Look Up Table method** is not used in case of PicoBlaze for reason of this processor resources. The LUT method makes possible the implementation of any logical function, in special case bijective.

The **LFSR (Linear Feedback Shift Register)** method is an alternative solution that provides assurance of bijectivity for such instructions as all the SHIFTs, if the characteristic polynomial

of LFSR is irreducible **(Hlawiczka 1997)**. In addition to bijectivity assurance, the LSFR method prevents faults masking, and allows cyclic usage of test vectors (irreducible polynomial). The LFSR method brings significant improvement in test results **(Wegrzyn 2014 B)**. Therefore, it is described in detail in chapter 6.3.

Choice of method of bijectivity assurance and detailed description of each bijective block construction is presented in chapter 5.4.

5.4. Composition of the bijective PicoBlaze test program

The PicoBlaze instructions which preserve bijective property by themselves every time or it is easy to realize bijective function of input arguments (test vectors) using them are: XOR, AND, OR, RR, RL, LOAD, STORE, COMPARE, TEST, FETCH, INPUT, OUTPUT, (JUMP, CALL)*. The PicoBlaze instructions which do not preserve bijective property by themselves are: ADD, ADDCY, SL0, SL1, SLA, SLX, SR0, SR1, SRA, SRX, SUB, SUBCY.

This part of the dissertation classifies methods of bijectivity assurance, and describes in details how the bijective property is assured for every individual instruction. The operation of every instruction described in PicoBlaze user manual is available on attached CD.

The **„XOR"** instruction preserves bijective property every time by itself. To check correctness of XOR operation, it is enough to execute XOR once with all bits equal „0" and then „1'', and do the same on two registers instead of the immediate values 0x00 and 0xFF. The test of this instruction can be implemented in the following way (see listing 5.4.1):

```
XOR   s7,   FF;   result of „XORing" s7 with 0xFF is placed in s7
XOR   s7,   00;
LOAD  s8,   FF;
XOR   s7,   s8;   result of „XORing" s7 with s8 is placed in s7
LOAD  s6,   00;
XOR   s7,   s6;   result of „XORing" s7 with s6 is placed in s7
```

Listing 5.4.1: „XOR" instruction testing

Moreover the **„XOR"** instruction is tested more times indirectly, as auxiliary instruction inside blocks to other instruction testing or as mentioned before bonding instruction.

The „**AND”** instruction may be bijective if I multiply any register by other filled in only „1” or when I multiply any registers by immediate value 0xFF. Thus, I have transformed AND instruction to satisfy **identity** data relation (see Listing 5.4.2). In every other case I observe loss of information so non-bijective behaviour. Of course I test this instruction in this way that it clears bits inside blocks to testing of other instructions too. So the instruction is fully tested both in direct and indirect way. The test of the „AND” instruction may be as follows:

```
LOAD  s6, FF;
AND   sD, s6;
AND   sD, FF;
```

Listing 5.4.2: „AND” instruction testing

The **„OR”** instruction inversely, can be bijective if it is executed upon register with value 0x00 or with the some register or a register which contains value 0x00. Thus, I have achieved here **identity** data relation in slightly different way. In other way, I could disturb bijectivity of them. OR instruction is tested indirectly in blocks to test other instruction. There it combines for instance information recovered from CARRY flag with content of given register. The OR test is realized as in Listing 5.4.3 and in Figure 5.4.3.

```
OR      sD,  00;   bitwise OR contents of SD register with literal 00
OR      sD,  sD;   bitwise OR contents of SD register with itself
LOAD    s3,  00;   load literal 00 into S3 register
OR      s3,  sD;   bitwise OR contents of SD register with SD register
```

Listing 5.4.3: „OR” instruction testing

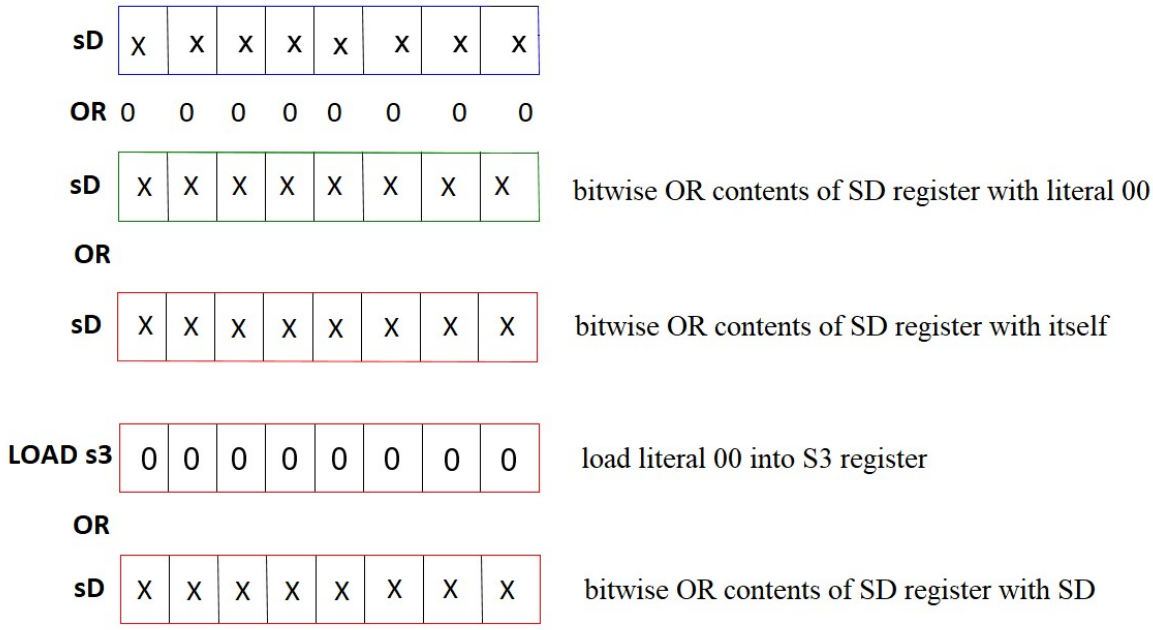


Figure 5.4.3: „OR" instruction testing

Such instructions as **LOAD, STORE, FETCH, INPUT, OUTPUT** are bijective **(identity)** by nature. If they work correctly, they cannot disturb bijective flow of data.

The **„LOAD"** instruction preserves alone the bijective transformation on the basis of **identity**. First the content of initial register sD is saved in sA register and then the content of sA register is negated (complemented) by execution of „XOR" instruction. Finally I reload saved initial content from register sA to sD (see Listing 5.4.4).

```
LOAD sA,  sD;  loading contents of sD register into sA register
XOR  sD,  FF;  obtaining opposite contents of sD register by XOR instruction
LOAD sD,  sA;  reload contents of  initial sD register.
```

Listing 5.4.4: „LOAD" instruction testing

The **„STORE"** and **„FETCH"** instructions, alone preserve the bijective transformation on the basis of **identity too**. They can be tested together in one block. Firstly I had saved content of initial register sD to scratchpad RAM and then I complemented the content by execution of „XOR" instruction. Finally I have reload saved initial content from the scratchpad to sD (see Listing 5.4.5).

```
STORE   sD,  05;  store register SD to scratchpad RAM location
XOR     sD,  FF;  obtaining opposite contents of SD register by XOR instr.
FETCH   sD,  05;  read scratchpad RAM location 05 into register SD (restore  SD reg).
```

Listing 5.4.5: „STORE, FETCH" instruction testing

The above block is written merely for testing **„STORE"** and **„FETCH"** instructions. Testing entire memories (write/read from different addresses) is out of my scope. For memory testing, i.e. „march" algorithms are intended **(Bushnell 2000)**.

There is a different situation when we take into consideration instructions as „**COMPARE"** and „**TEST"**. These instructions do not even have possibility of interference of bijective data flow, because they are designed exclusively to set flags, but not to modify data upon they work. It works obviously if these instructions operate correctly, and the data remain **identical**.

The **„COMPARE"** instruction performs an 8-bit subtraction of two operands but only affects the ZERO and CARRY flags. For instance the „JUMP" instruction is sensitive to the flags, so I can check correctness of „COMPARE" instruction behaviour by checking the flags (as in Listing 5.4.6). When zero flag is set, content of register is erased. It should be noticed, that the zero flag is set exclusively for one test vector equal 0x80. Faulty behaviour of the „COMPARE" instruction would manifest in wrong results from output of processor.

```
      COMPARE  sD,  80;  set ZERO flag to MSB of sD register
      JUMP     nz,  lx;  jump if the ZERO flag is not set
      AND      sD,  00;  AND with „0" is executed when the ZERO flag is set
Lx:   ADDCY    sD,  00;  if ZERO flag is set „1" and CARRY flag is set „0" , trial
                     ;   combining probable carry with initial register
```

Listing 5.4.6: „COMPARE" instruction testing

The **„TEST”** instruction performs bit testing via a bitwise logical ,,AND” operation between two operands. Unlike the „AND” instruction, only the „ZERO” and ,,CARRY” are affected; no registers are modified. Thus behaviour of the instruction may be checked by checking ZERO and CARRY flags. In the following example the ZERO flag can be set only for one test vector 0x80. In the case when TEST set the ZERO flag incorrectly, additional ADD instruction would be executed. It would manifest in wrong execution results. If the carry flag is set incorrectly, it would detect by ADDCY instruction and results of the program execution would be different as default (see Listing 5.4.7).

```
      JUMP    tst;       test sX, kk
zer:  ADD     sD,  01;
      JUMP    8z;
tst:  TEST    sD,  80;   set ZERO flag to MSB of SD register
      JUMP    z,   zer;  if ZERO flag is set JUMP to zer label
      ADDCY   sD,  00;   combining probable carry with initial register
     ..................................
  8z: (next block of program)
```

Listing 5.4.7: ,,TEST” instruction testing

Another situation is in the case of ,,**JUMP”** instructions. These are sensitive to flags or are unconditional. These do not work upon data and cannot modify it directly. So I can't even consider if they preserve bijective property or not. However if ,,JUMP” instructions work incorrectly, they can change data flow and one or more blocks of program could operate completely different as it was intended, and of course cause loss of bijectivity. So only faulty ,,JUMP” instruction operation can disturb bijective flow of determined test program.

At the **„ADD”** instruction it is easy to ensure bijective property by **continuous addition** of **„1s”** to a register (see Listing 5.4.8). When the addition is correct and the results less or equal to 255, bijectivity is preserved automatically. When CARRY occurs I handle this situation programmatically and combine the CARRY bit into the outcome. In order to test **„ADD”** instruction both general-purpose register Sx result, and ZERO and CARRY flags should be tested. For this purpose, I propose the following code:

```
15z:      JUMP  addins;     jump to beginning add test
lbc:      ADD   s7,  01;    s7->s7
          JUMP  16z;        jump to the label 16z (Next instruction test)
addins:   ADD   s7,  ff ;   adding number 0xff to the content of register s7
          JUMP  z,   adcy;  checking if ZERO is set correctly
          JUMP  nc,  lbc;   jump to the label lbc, and checking if CARRY is set correctly
adcy:     ADDCY s7,  00;    adding number 0x00 and carry bit to content of register s7
16z:      Next_instruction_test: ....
```

Listing 5.4.8: „ADD" instruction testing

I resigned from application with two registers arguments. This could lead to 2^{16} (65 536) input vectors instead of 256 and obviously to excessively long time of simulation. Any way, if the previous block is bijective, all possible numbers generated before appear. This allows me to get all possible results.

Similar situation is regarding „**ADDCY"** (ADD with CARRY) instruction. The only difference is that the CARRY flag should be set occasionally by the preceding instruction in order to check if „ADDCY" adds correctly the flag to its result of execution. In another case „ADDCY" – cannot be bijective by itself. To sum up, I have achieved bijectivity of „ADDCY" instruction by **continuous addition of „1s" and by generating of input CARRY.**

Taking into consideration „**SUB"** instruction, I followed analogical as in case of „ADD". I achieved the bijective property applying **continuous subtraction of „1".** „SUB" can be bijective until I do not exceed the subtraction range. Then I have to handle such a situation by the dedicated subroutine in which I have to check if the CARRY flag was set correctly this time. I used the instruction „SUBCY" for this purpose which capture CARRY bit set by SUB. I proposed following code to SUB instruction testing (see Listing 5.4.9):

```
          JUMP   subins;    SUB test
bsc:      SUB    s7,  01;   result of subtraction of s7 – 0x01 placed in register s7
          JUMP   15z;       label at block to another instruction testing
subins:   SUB    s7,  F7;   result of subtraction of s7 – 0xF7 placed in register s7
          JUMP   nc,  bsc;  jump to the label bsc if no CARRY
          SUBCY  s7,  00;   capturing of CARRY flag at subtraction
15z:      Next_instruction_test: ....
```

Listing 5.4.9: „SUB" instruction testing

In the case of „**SUBCY"** (Subtract with CARRY) instruction there is an additional need to check if a CARRY flag set by the preceding instruction was included in the calculation of the difference from subtraction. To **generate the CARRY** I utilized „COMPARE" instruction between a register with input data to the program block and constant value. The main principle is the same as for „SUB"; **continuous subtraction of „1s".**

Instructions **„SRA"** (Shift Right through All bits, including CARRY), **„SLA"** (Shift Left trough All bits, including CARRY) are not bijective by themselves. In order to assure bijectivity I need to **generate the CARRY flag** using for instance „TEST" instruction. Admittedly „SLA" generates CARRY flag at its execution, but then only later at next execution can capture back the same CARRY generated by itself. But such a cycle is too long to assure bijectivity. There appears lack of information on MSB or LSB respectively if CARRY is not generated before. Thus I had to prepare CARRY flag before the first execution of „SRA" or „SLA". In this way concurrently LSB, (MSB) is copied to CARRY and MSB, (LSB) is filled with earlier value from CARRY. Obviously I may check additionally if „SLA" instruction sets CARRY flag properly executing example code after dashed line (in Listing 5.4.10):

```
LOAD sC,  s0;
TEST sC,  80;  set carry flag to 7th bit of SC register (enough to assure bijectivity)
SLA  sC;       shift left arithmetic sC register
-------------------------------------------------------------------------
ADDCY sF, 00;  restore MSB bit of  initial register
OR    sC, sF;  combining restored MSB bit with shifted
AND   sF, sC;  value without loss of  information
AND   sC, FE;  the information about MSB is preserved in SC register
          ;    ( initial value can be restored )
```

Listing 5.4.10: „SLA" instruction testing

„RR" (Rotate Right), **„RL"** (Rotate Left) can be bijective themselves. These do not lose the information. Here appears only the need to check if they set CARRY flag correctly. So the flag is captured in the CARRY after execution of this instruction, and is combined with a value of shifted register. The flag has influence on final result too. I have examined „RR" instruction as follows (see Listings 5.4.11, 5.4.12):

```
LOAD    s9,  00;
RR      sC;       set carry flag of SC register to LSB bit
ADDCY   s9,  00;  store LSB of sC  register in s9 register
XOR     sC,  s9;  combine restored LSB bit with shifted value without loss
                  ; of information into sC register
```

Listing 5.4.11: „RR" instruction testing

and respectively RL:

```
LOAD    s0,  00;
RL      sF;       set carry flag of sF register to MSB bit
ADDCY   s0,  00;  store MSB of sF  register in s0 register
RR      s0;       rotate bit 0 with information about CARRY to MSB position
XOR     sF,  s0;  combine restored MSB bit with shifted value without loss
                  ; of information into sF register
```

Listing 5.4.12: „RL" instruction testing

The next instructions **„SR0"** (Shift Right with „0" fill), **„SR1"** (Shift Right with „1" fill) which do not preserve themselves the bijective transformation due to the fact that MSB is affected by „0'' or „1'' respectively, regardless of input data. The instructions Shift Right with „0" (SR0) or „1" fill (SR1) stores the LSB in CARRY, „0" or „1'' is filled on the position of MSB and others bits are shifted by one position to the right. The CARRY is captured in sE register and then it is rotated as in earlier examples. I need erase „1" from MSB position after „SR1" is executed. It is realized by execution of AND instruction between sA register and immediately value 7F. Finally by replacing it with bit 0^{th} full flow of information is guaranteed. Thus full bijectivity is assured (see listing 5.4.13 and Figure 5.4.13).

```
LOAD    sE,  00;
LOAD    sA,  sF;
SR1     sA;       shift register SA right, „1" fill on bit 7th (MSB)  position
ADDCY   sE,  00;  restore LSB of initial register in SE register
RR      sE;       rotate captured value  in SE
AND     sA,  7F;  mask of sA register
OR      sA,  sE;  combining restored LSB bit with initial register
```

Listing 5.4.13: „SR1" instruction testing

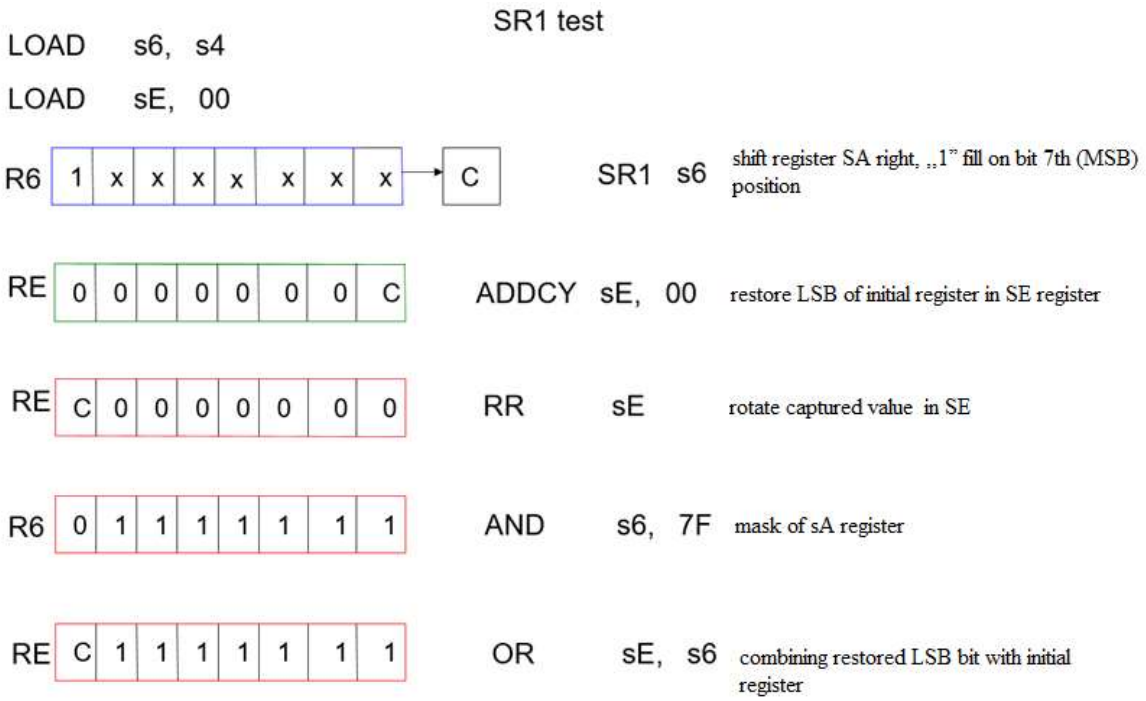


Figure 5.4.13: „SR1" instruction testing

„SL0" (Shift Left with „0" fill), **„SL1"** (Shift Left with „1" fill) instructions are not bijective for the similar reason as „SR0" and „SR1", because LSB is affected by „0", „1" respectively, independently of input data. I applied the LUT method to solve this problem too. „SL0", „SL1" sends MSB to CARRY. After execution of „SL0" it is enough to capture CARRY by execution ADDCY upon the same register. In case of „SL1" I need catch CARRY to different register, erase „1" from the position of LSB (AND FE). Then to replace value on LSB by „OR".

Instruction **„SLX"** (Shift Left eXtend bit „0") does not preserve alone bijective property, because on positions of bit 1^{th} and bit 0^{th} are constantly the same values (from bit 0^{th}), and in this way information is lost. The method of **XORing data** proved to be sufficient in this case. MSB is sent out to CARRY flag. The carry flag is captured to s3 register. In order to implement a bijective transformation the value of carry flag captured to register s3 is combined with content of register sE by „XOR" instruction (see Listing 5.4.14):

```
SLX     sE;       shift register sE left. Bit 0th (LSB) is unaffected. MSB is sent to carry
ADDCY   s3,  00;  restore MSB of initial register in S3 register
XOR     sE,  s3;  combining restored MSB with initial register
```

Listing 5.4.14: „SLX" instruction testing

Following example presents testing of **„SRX"** (Shift Right, sign eXtend, Bit 7 is unaffected) instruction. Method based on **XORing data** to implement a bijective transformation is almost the same as this applied to „SLX". The only difference is that the same values are constantly on the position of 7th and 6th. LSB is sent to CARRY. The CARRY is captured by executing next instruction „ADDCY" and is located on LSB in s2 register. In order to move it to MSB location RR (Rotate Right) instruction is executed upon s2 register. Then CARRY is combined into data vector by executing XOR instruction (as in Listing 5.4.15):

```
SRX     s0;       shift register S0 right. MSB is unaffected and extended on 6th bit
        ;         position. LSB is sent to carry
ADDCY   s2, 00;   restore LSB of initial register in s2 register
RR      s2;       rotate right captured value in order to obtain LSB on MSB position
XOR     s0, s2;   combining restored MSB with initial register
```

Listing 5.4.15: „SRX" instruction testing

5.4.1. Experimental results of bijective merely program

Some comparisons of the fault coverage achieved by deterministic test programs with the fault coverage achieved by well-known programming algorithms were made in the bibliography **(Corno 2003)**. I have implemented some other programs that exploit the functionality of the processor core as: data sensitive path initial approach **(Wegrzyn 2007 B)**, conventional implementation of Fibonacci series or its recursive version, the application specific programs **(Wegrzyn 2009),** three matrix multiplication applications **(Wegrzyn 2014 A)**. Composition of the program written according to principle of data sensitive path **(Wegrzyn 2007 B**) in form of one big block of instructions turned out to be difficult. This program is not completely bijective. A large number of data sensitive paths could not be activated in this way, thus it achieved medium FC 70,2%. The Fibonacci recursion exploits additionally such resources of the processor as i.e. stack. The matrix multiplication applications employ significantly fewer assembler instructions, and achieved by them FC is clearly smaller. The researches results have proved, that the data-sensitive path and bijective property provide

a way of randomizing the instruction operands of the test sequence, resulting in increased processor activity and consequently in increased fault coverage.

The results are given in Table 5.4.1. The achieved fault coverage is presented in two column pairs **(Wegrzyn 2009)**. The first column pair refers to the fault coverage (FC) where only stuck-at faults were injected (see the chapter 8.3). The second column pair presents the complete fault coverage with both stuck-at faults and functional faults in the LUTs. For each pair, the left-hand column refers to all the simulated faults (i.e. 934 in the case of the stuck-at faults and 1804 in the case of the stuck-at faults and the SEU faults in the LUTs). The test sequence does not test some specific types of faults related to input/output operations (e.g. interrupt driven routines). If I neglect these faults, the fault coverage of the remaining faults (i.e. 883 in the case of the stuck-at faults and 1603 in the case of the stuck-at faults and the functional faults in the LUTs) is given in the right-hand column.

Table 5.4.1: My Solution-Sensitive-Path Approach: Obtained Initial Fault Coverage

test program	FC (%) - Stuck-at faults		FC (%) - complete list	
	934 faults	883 faults	1804 faults	1603 faults
Fibonacci	49.9	52.8	33.8	37.7
Fibonacci (recursion)	56.0	59.2	43.3	48.4
MM1	63.6	67.3	49.9	56.1
MM2	64.4	68.1	51.2	57.6
MM3	66.4	70.2	53.1	59.7
random instruction order	62.7	66.4	46.8	52.3
data sensitive path (not completely bijective)	72.6	76.8	62.9	70.2
data sensitive path + bijective sub-sequences	88.1	93.2	76.6	85.6

The fault coverage for the stuck-at faults is given only for a comparison with other reported solutions. The main interest is focused on the fault coverage of the complete list of the injected faults. The initially achieved fault coverage for the proposed approach was 76.6%. If I neglect the faults related to the input/output operations I get an 85.6% fault coverage.

6. Optimal reduction of number of test vectors

One of the most important criterion of every test program evaluation are Fault Coverage (FC) and time required to complete. The test time has influence on costs of production. This time depends on both number of program instructions to be executed and number of applied test vectors. This chapter focuses on optimization of the number of test vectors with as low as possible influence on FC. Such approaches may be especially profitable in the case of testing 32 or 64-bit microprocessors as there is a huge number of input test vectors required for exhaustive testing of these microprocessors. For instance there is 2^{32} possible input test vectors for a 32-bit microprocessor.

Two goals are taken under consideration in this chapter. The superior objective is to find a minimal set of test vectors, which can achieve the maximal Fault Coverage (FCmax). This means, that developed optimization method should give the same FC as an exhaustive test. The second purpose is to obtain very high FC for a limited number of test vectors by optimization of test vectors set or determination of every next vector using several proposed algorithms. This method can be particularly useful in a case of non-exhaustive test, when for instance achievement of the 97% FCmax is satisfactory.

6.1. BIST implementation

After I have developed a test program, which achieved a good FC, the next step is optimization of a set of test vectors at BIST implementation. A common test strategy aims at minimizing test time (the number of test vectors) and maximize FC. These two goals are contradictory, as will be shown in the followings. Therefore the first attempt is to minimize the number of test vectors without decreasing the FC.

A general test situation can be described by:

- the set of faults $F = \{f_1, f_2, ..., f_m\}$;
- the set of available tests vectors $V = \{v_0, v_1, \ldots, v_{n-1}\}$ where v_i corresponds to the execution of the test sequence (program) with a binary input value i ($0 \le i \le n-1$); in my case of 8-bit microprocessor there are 256 binary input values
- the test matrix D describes test capabilities of tests T (detect-ability of faults f_j by test vector v_i). Each test vector v_i, $0 \le i \le n-1$, is related to a binary test vector $d_{ji} = [d_{0i}, d_{1i}, \ldots, d_{mi}]$. The value $d_{ji} = 1$ denotes that test vector v_i detects fault f_j. Conversely, $d_{ji} = 0$ indicates that v_i does not detect f_j. Binary test matrix D consists of test vectors $D = [d_0, d_2, \ldots, d_{n-1}]$. Its dimension is $m \times n$. In my particular case, $m = 1603$, $n = 256$.

In order to better understand optimization of the number of test vectors, I proposed the following definitions:

Definition 6.1: *The fault of i-th order is called a fault detected exclusively by i test vectors.*

Definition 6.2: *The vector of i-th order is called a vector, which detects at least one fault of i-th order and does not detect any fault of lower order than i.*

Consequently, the most difficult to detect faults, called further the hardest faults are the first-order faults, which are detected only by one test vector. In my practical case I have found 41 faults detected only by one vector. Table 6.1 presents statistics on faults order. Faults of these orders are present in outcome file from the fault simulation experiment.

Table 6.1: Statistic of faults orders

Order	1	15	67	71	72	78	80	87	92	99	125	127	128
# faults	41	1	1	1	1	1	1	1	1	1	1	2	110
Order	135	144	160	161	168	175	176	184	190	191	192	193	194
# faults	1	1	2	1	2	1	3	6	1	4	32	2	3
Order	195	196	199	200	204	206	208	209	215	216	217	219	222
# faults	2	1	1	4	1	1	4	1	1	4	1	1	1
Order	223	224	225	226	227	230	231	232	234	235	236	239	240
# faults	1	34	4	8	3	2	1	2	4	2	1	2	103
Order	241	242	243	248	249	250	251	252	253	254	255	256	
# faults	19	2	2	31	2	3	2	6	3	10	146	731	

There are 41 faults of first order, one fault of 15-th order, one fault of 67-th order, and so on. There is the highest number of 256-th order faults (731). It is worth of notice, that faults of higher-orders are usually covered by first order vectors. By definition, it holds for 256-th order faults. Experiments proved that this holds also for the 15-th order faults. The 15-th order fault is covered by at least one first-order vector. In case of the 67-th order faults, 4 from 67 vectors are first order, and so on. For example fault (217), 15-order, is detected by the vectors: 7D, 6D, 5D, 4D, BE, AE, 9E, 8E, EE, DE, CE, 3D, 2D, 1D, 0D. It turned out, that all these vectors are 1-st order.

Fault (1030), 67-th order, is detected by vectors: 7E, 3E, 3D, B7, 39, BB, 35, AF, 31, 2D, 0D, ...,03.

Four these vectors are first order. Above statistics gives us information how efficient is bijective test program. Hence, if the number of low orders faults was high, and high number of the lowest-order vectors was required to detect them, it would mean that small number of sensitivity paths were activated. Then it seems a good idea to improve the test program (written in PicoBlaze assembler, see chapter 5). In another case, one can expect good results by optimizing the set of test vectors applying the algorithms proposed bellow.

6.1.1. Algorithm 1 – greedy algorithm

An algorithm have been taken up in order to minimize the set of test vectors required to obtain maximal fault coverage (FCmax). This greedy algorithm selects first the best vector, i.e. a vector which covers the largest number of faults.

Algorithm 1: Greedy algorithm: the vectors that detect the largest number of faults first

Determine set F of all faults f_i
While F is not empty
{ Determine test vector v_i which covers maximum number of faults in set F;
Test the microprocessor with test vector v_i;
Remove all faults f_j that are covered by test vector v_i from set F;
}

In case of Algorithm 1 and penultimate version of my bijective program, the set of 33 such vectors appears enough to reach FCmax of 85.6% (see Table 6.2). This experiment showed that application of all 256 tests vectors (exhaustive test) for the PicoBlaze processor is redundant. Moreover, it turns out, that I can shorten about eight times the exhaustive testing time. It has crucial meaning in case of 32-bit or 64-bit microcontrollers, where application of all possible 2^{32} or 2^{64} test vectors becomes impossible for reason of testing time.

Table 6.2: Aggregation of FC for Algorithm1

#	Vector	Aggregated FC	% FCmax	Number newly detected faults by the vector
1	B1	1255	91,47	1255
2	4E	1322	96,36	67
3	7D	1338	97,52	16
4	8E	1342	97,81	4
5	7F	1344	97,96	2
6	F7	1345	98,03	1
7	EF	1346	98,10	1
8	F3	1347	98,18	1
9	5D	1348	98,25	1
10	4D	1349	98,32	1
11	AE	1350	98,40	1
12	9E	1351	98,47	1
13	F6	1352	98,54	1
14	EE	1353	98,62	1
15	DE	1354	98,69	1
16	CE	1355	98,76	1
17	3D	1356	98,83	1
18	2D	1357	98,91	1
19	1D	1358	98,98	1
20	0D	1359	99,05	1
21	77	1360	99,13	1
22	F5	1361	99,20	1
23	F1	1362	99,27	1
24	F8	1363	99,34	1
25	FC	1364	99,42	1
26	F0	1365	99,49	1
27	F4	1366	99,56	1
28	FB	1367	99,64	1
29	6D	1368	99,71	1
30	FA	1369	99,78	1
31	F2	1370	99,85	1
32	F9	1371	99,93	1
33	BE	1372	100,00	1

From Table 6.2, the most important observation is, that the speed of achieving FCmax decreases rapidly, and from iteration 6 and above only a single fault is covered by an iteration. The next algorithm is derived from this phenomenon.

6.1.2. Algorithm 2 – the lowest-order vector first

In order to minimize the number of test vectors without decreasing the FC, I developed Algorithm 2 according to which the test program selects the lowest-order test vectors first.

Algorithm 2: the lowest-order vector first

Determine set F of all faults f_i;
While *F is not empty*
{ Select the lowest order fault f_i of set F;
Select a test vector v_i that detects fault f_i;
Test the microprocessor with test vector v_i;
Remove all faults f_j covered by test vector v_i from set F;
}

In the implementation of the above algorithm, 28 vectors are enough to obtain maximal fault coverage FCmax. One of the vectors (7D), has detected 12 first-order faults. Vector 7E has detected 3 of first-order faults. The other 26 vectors have detected only one first-order faults each (see Table 6.3).

Table 6.3: List of the first-order vectors

# iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Test vector	7D	7E	FB	FA	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0
# detected 1-st order faults	12	3	1	1	1	1	1	1	1	1	1	1	1	1
# iteration	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Test vector	EE	DE	CE	BE	AE	9E	FC	6D	5D	4D	3D	2D	1D	77
#detected 1-st order faults	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Initially it might seem that greedy algorithm (Algorithm 1) should give a better result (lower number of test vectors) than Algorithm 2. However, it is not the case, Algorithm 2 results in 28 test vectors, in comparison to 33 test vectors for Algorithm 1. After thorough consideration, it is obvious

that starting with first-order test vectors gives the optimal solution as in order to obtain FCmax all first-order vectors must be tested. This can be easily derived from Definition 6.1 and 6.2.

Nevertheless it is not obvious, how the Algorithm 2 should be constructed after all first-order vectors have been tested. Fortunately in my case, testing all first-order vectors is enough to cover all higher order faults. Based on statistics of fault orders and further experiments, I can conclude, that most of higher order faults are covered by many vectors from the set of the first order. Hence, I may propose the method of testing the processor consequently with increased order vectors, until all faults are covered. Nevertheless in general case, Algorithm 2 should be improved. When there are two or more faults of the same lowest order greater than first order, several different vectors can be taken. In this case, the vector which covers the largest number of faults is selected. Consequently, improved algorithm, Algorithm 3 - Hybrid, is a mixture of Algorithm 2 and 1, however Algorithm 2 is higher priority algorithm. Algorithm 3 is especially useful in the case when the lowest-order vector is a second or larger order one, as in this case there are two or more vectors that cover the same fault. A proposition of the improvement is given below. It should be noted that performance of Algorithm 3 was not tested in practice as in this experiment testing only first-order vectors result in FCmax.

Algorithm 3: Hybrid (improved the lowest-order vector first)
Determine set F of all faults f_i;
While *F is not empty*
{ Determine subset F_i of F with the same, lowest-order faults f_i
Select a vector v_i that detects the largest number of faults from set F_i
test the microprocessor with test vector v_i;
from set F, remove all faults f_j that are covered by test vector v_i;
}

The best results has achieved ,,Hybrid" algorithm. Its simplified version Algorithm 2 – ,,The lowest order vector first" is characterized by uneven increments of FC, but it is simpler and requires slightly less calculation time. Algorithm 1 - ,,Greedy" requires higher number of iterations to achieve FCmax. Algorithm 3 ,,Hybrid" turned out to be the best in my practical case.

6.2. Number of iteration required to achieve 97% FCmax

As mentioned at the beginning of chapter 6 a Fault Coverage at the level of 95 or 97% of FCmax may be satisfactory at practice. For this reasons I have checked, how quickly the FC tends to its maximum value, applying presented before algorithms.

6.2.1. Number of iteration for Algorithm 1

The initial goal was to reach FCmax with the lowest number of vectors (the result was 33 vectors), which is larger than for Algorithm 2 (28 vectors). However, the greedy algorithm (Algorithm 1) results in faster increase of FC for initial iterations. On the other hand, this algorithm generates five redundant vectors to obtain FCmax. This phenomenon was discovered at trial of generation of all permutations of 32 remained vectors except the best one (32!). The aggregation of the FC is presented in Table 6.2, and repeated in Table 6.4. Only 3 iterations are required to obtain 97% FCmax.

Table 6.4: Aggregation of fault coverage to FCmax

#	Vector	Aggregated FC	% FCmax	Number of newly detected faults by the vector
1	B1	1255	91,47	1255
2	4E	1322	96,36	67
3	7D	1338	97,52	16
4	8E	1342	97,81	4

6.2.2. Number of iteration for Algorithm 2

Algorithm 2 has used 26 -test vectors which detect only a single first-order faults each, one vector which detects 3 faults of first order, and one vector which detects 12 faults of first order. Additionally, the FC achieved when every of these vector was applied alone was calculated. Based on these results it is possible to determine the order of applied test vectors. This order had been determined once, before

Algorithm 2 started. However this approach allows to achieve FCmax, but the number of iterations required to achieve 97% of FCmax is optimized further in chapter 6.2.3.
Table 6.5 presents the Fault Coverage achieved by an individual vector of first order alone. The result file from fault simulation is the same as for data presented in Tables 6.2 and 6.3.

Table 6.5: Fault Coverage achieved by individual vector of first order

Vector	F2	F0	F4	F6	F1	FA	F8	EE	CE	2D
Detected faults	1249	1240	1240	1236	1231	1229	1227	1226	1221	1220
%FC max	91,03	90,38	90,38	90,09	89,72	89,58	89,43	89,36	88,99	88,92
Vector	FC	F5	F3	77	1D	DE	F9	F7	AE	3D
Detected faults	1220	1218	1217	1214	1209	1205	1203	1202	1198	1193
%FCmax	88,92	88,78	88,70	88,48	88,12	87,83	87,68	87,61	87,32	86,95
Vector	9E	FB	4D	BE	5D	6D	7E	7D		
Detected faults	1187	1185	1184	1175	1164	1157	1149	1099		
%FCmax	86,52	86,37	86,30	85,64	84,84	84,33	83,75	80,10		

The aggregation of detected faults was carried out. The results are presented in Table 6.6. It is visible, that it is enough to apply 11 vectors of first order for achieving 97% of FCmax. These vectors are ordered by its fault coverage presented in Table 6.5.

Table 6.6: Aggregation of FC for Algorithm 2

#	Vector of 1-st order	AggregatedFC	% FCmax	Number of newly detected faults by the vector
1	F2	1249	91,03	1249
2	F0	1270	92,57	21
3	F4	1277	93,08	7
4	F6	1278	93,15	1
5	F1	1286	93,73	8
6	FA	1295	94,39	9
7	F8	1296	94,46	1
8	EE	1310	95,48	4
9	CE	1320	96,21	10
10	2D	1321	96,28	1
11	FC	1337	97,45	16
12	F5	1338	97,52	1
13	F3	1339	97,59	1
14	77	1340	97,67	1
15	1D	1342	97,81	2
16	DE	1343	97,89	1
17	F9	1344	97,96	1
18	F7	1345	98,03	1
19	AE	1346	98,10	1
20	3D	1347	98,18	1
21	9E	1348	98,25	1
22	FB	1349	98,32	1
23	4D	1350	98,40	1
24	BE	1354	98,69	4
25	5D	1355	98,76	1
26	6D	1356	98,83	1
27	7E	1359	99,05	3
28	7D	1372	100,00	13

6.2.3. Number of iteration for Algorithm 3 - Hybrid

In Table 6.6 only first-order vectors are used. In the case when all 28 first-order vectors are tested (FCmax is obtained), the order of vectors is not important so Algorithm 2 is simpler one. However for reduced number of test vectors, the speed of FC is very important, so the vectors order should be improved. In this case Algorithm 3 – Hybrid should be employed, this algorithm is an improved version of Algorithm 2. Algorithm 3 obtains 97% FCmax in 4 iterations. The results of this experiment are presented in Table 6.7.

Table 6.7: Aggregation of FC for Algorithm 3

#	Vector of 1-st order	Aggregation of detected faults	% FCmax	Number of newly detected faults by the vector
1	F2	1249	91,03	1249
2	1D	1316	95,92	67
3	7D	1330	96,94	14
4	F7	1343	97,89	13
5	7E	1347	98,18	4
6	FB	1350	98,40	3
7	F3	1351	98,47	1
8	6D	1352	98,54	1
9	5D	1353	98,62	1
10	4D	1354	98,69	1
11	BE	1355	98,76	1
12	AE	1356	98,83	1
13	9E	1357	98,91	1
14	FA	1358	98,98	1
15	F6	1359	99,05	1
16	F0	1360	99,13	1
17	DE	1361	99,20	1
18	CE	1362	99,27	1
19	EE	1363	99,34	1
20	3D	1364	99,42	1
21	2D	1365	99,49	1
22	77	1366	99,56	1
23	F9	1367	99,64	1
24	F5	1368	99,71	1
25	F1	1369	99,78	1
26	F8	1370	99,85	1
27	FC	1371	99,93	1
28	F4	1372	100,00	1

The comparison of the aggregated FC for all three Algorithms is shown in Table 6.8 and in Figure 6.1. It can be seen that Algorithm 1 results in the highest FC only for three initial iterations. Higher FC achieves Algorithm 3 since 4th iteration. Using Algorithm 2 FC increased irregularly. This algorithm is not optimal, because decision about choice of a vector for the next iteration was made on base of the highest number of covered faults by individual vector at the very beginning of the algorithm.

Table 6.8: The comparison of the FC aggregation for three proposed algorithms

#	Algorithm 1 %FCmax	Algorithm 2 %FCmax	Algorithm 3 %FCmax
1	91,47	91,03	91,03
2	96,36	92,57	95,92
3	97,52	93,08	96,94
4	97,81	93,15	97,89
5	97,96	93,73	98,18
6	98,03	94,39	98,40
7	98,10	94,46	98,47
8	98,18	95,48	98,54
9	98,25	96,21	98,62
10	98,32	96,28	98,69
11	98,40	97,45	98,76
12	98,47	97,52	98,83
13	98,54	97,59	98,91
14	98,62	97,67	98,98
15	98,69	97,81	99,05
16	98,76	97,89	99,13
17	98,83	97,96	99,20
18	98,91	98,03	99,27
19	98,98	98,10	99,34
20	99,05	98,18	99,42
21	99,13	98,25	99,49
22	99,20	98,32	99,56
23	99,27	98,40	99,64
24	99,34	98,69	99,71
25	99,42	98,76	99,78
26	99,49	98,83	99,85
27	99,56	99,05	99,93
28	99,64	100,00	100,00
29	99,71		
30	99,78		
31	99,85		
32	99,93		
33	100,00		

However this algorithm is easier to implementation and quicker to execution than Algorithm 3. For Hybrid Algorithm, the number of covered faults are calculated at every iteration of the algorithm. Algorithm 1 „Greedy" can be used when the rapid increase of FC in the first few iterations is required. On the other hand, this algorithm requires the highest number of iterations (33) to complete (obtain FCmax). It is predictable, that for other sets of input data, the difference in the number of iterations may be even greater in favor of Algorithm 3 „Hybrid''. One of the most important observations is, that for Algorithm 1 the number of iterations to achieve FCmax depends more on correlation between vectors which detect faults of first order (determined in chapter 6.1) and vectors which detects the highest number of faults. Taking into consideration both number of iteration required to achieve FCmax and FC increase rate, it is possible to conclude that Algorithm 3 is the best. However more sophisticated algorithms i.e. genetic algorithm, simulated annealing might give better solutions. But such algorithms should be applied both for all test program generation together with optimization of test vector set. It was necessary to use all vectors of first-order to achieve FCmax, and I have reduced 9 times (256/28) numbers of test vectors required to achieve FCmax.

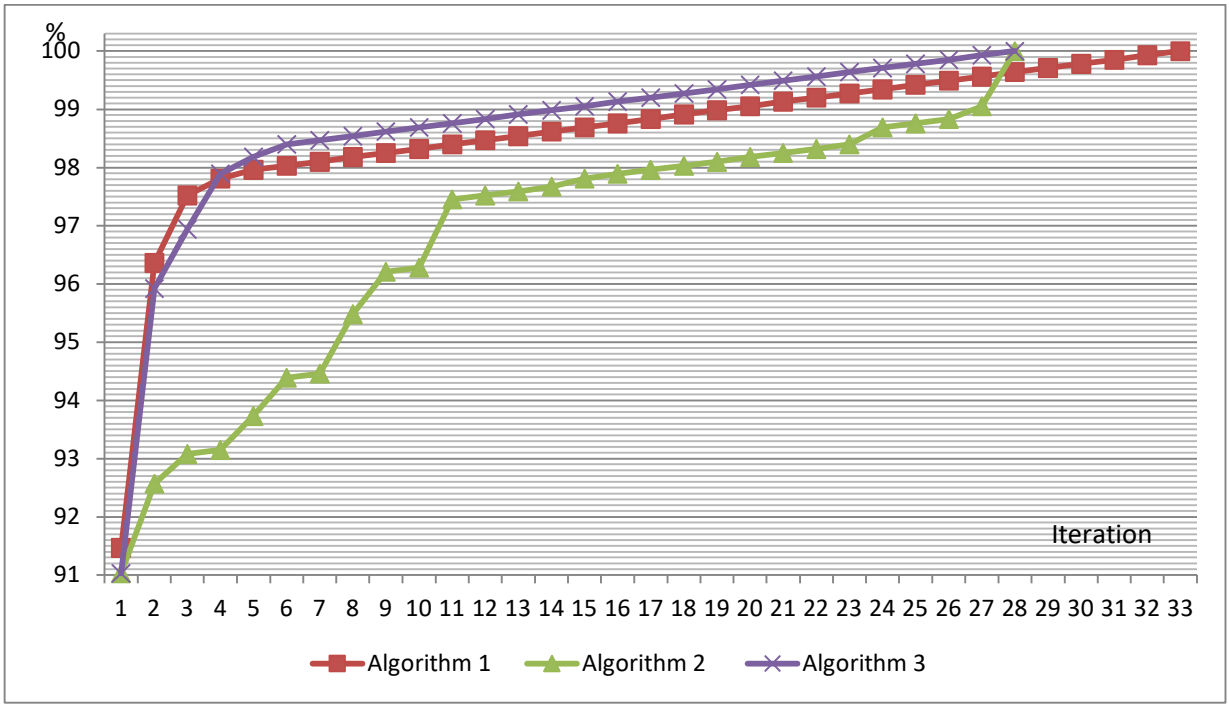


Figure 6.1: The comparison of the FC aggregation for all three Algorithms

6.2.4. Optimal reduction of number of individual blocks test vectors

After developing of optimization strategies for the complete test program, especially determination of optimal set of test vectors which achieves FCmax, so called further global test vectors, it seems interesting and beneficial to look deeper into processor blocks, individual functions of test program dedicated to individual processor block testing. The next intention is to determine optimal set of local vectors i.e. vectors for testing individual processor blocks and comparison how these vectors corresponds to the optimal set of global vectors.

PicoBlaze VHDL description is divided into 13 blocks by its author (Ken Chapman). I have determined optimal sets of test vectors separately for every hardware block of the processor, if it was possible. I utilized primarily the Algorithm 3 – Hybrid, and Algorithm 1 to this task. Algorithm 1 - Greedy I have applied usually in cases of blocks, where the lowest order of test vector was relatively high i.e. 67 or higher. Determined set of vectors achieves FCmax for every individual PicoBlaze block FCmax(block). Accordance to Chapman the PicoBlaze blocks together with proper sets of test vectors determined by myself are presented in Table 6.9. There exist a few such PicoBlaze HW blocks, where I cannot distinguish special function dedicated to test them. Such block are i.e. ZERO and CARRY Flags, Program Counter, etc. Every kind of instructions such as Logical, Arithmetical and Shifts have capability to generate flags. Every instruction of my bijective program tests PC, so for such a blocks determined before in this chapter global test vectors are applied.

As a result of these researches, a few new vectors appeared required to optimal test of individual blocks. These test vectors are determined on the basis of detailed analysis of software functions dedicated to testing every individual processor block, software functions which test individual processor blocks in indirect way, and first of all interaction between processor HW and specific assembler instructions which operate on its specific arguments. Examples of such analyses are presented in chapters 8 and 9. Short description of research results for individual processor blocks is following:

1. **Fundamental Control**

   Almost every vector is capable of achieving FCmax of this block. For instance I can apply global 1-st order one: F7.

2. **Interrupt input logic. Interrupt enable and shadow Flags**

   FCmax is obtained by e.g. global 1-st order vector F7 too.

3. **Decodes for the control of the program counter and CALL/RETURN stack**

There are not direct access to the Program Counter from assembler instructions level. CALL/RETURN stack is tested by dedicated assembler function based on CALL assembler instruction. This function consists of 31 assembler instructions. It is possible to determine vectors which test this block in indirect way by the used assembler function. Two from set of global 1-st order vectors: 7E (detects 1-st order fault in this block), F7 are enough to achieve FCmax of this block.

4. **The ZERO and CARRY Flags**

There are 29 faults of first order covered by 28 vectors. One from these vectors tests 2 faults of first order. So, FCmax of this block is covered by 28 vectors contained in Table 6.9. Only Hybrid Algorithm was applied for reason of high number of 1-st order vectors. This block is tested by all the bijective program (about 370 assembler instructions).

5. **The Program Counter. Definition of a 10-bit counter which can be loaded from two sources**

There are four faults of 1-st order tested by the same vector 7D. All faults in this block are covered by two global vectors of 1-st order 7D and F7. This block is tested by all the bijective program (about 370 assembler instructions).

6. **Register Bank and second operand selection**

There isn't any fault of first order in this block. By applying Algorithm-1 Greedy, only two vectors are required to achieve FCmax for this block. Whereas Algorithm-3 Hybrid requires 3 test vectors. This block is tested by all the bijective program (about 370 assembler instructions) too.

7. **Memory Storing Function Block** requires only one test vectors: F7 according to both algorithms.

8. **Logical Operations Block**

There is lack of 1-st order faults in this block. Applying Algorithm-3 – Hybrid, there are required 4 vectors for achievement FCmax(Logical) as presented in Table 6.9. When I have applied the same Hybrid Algorithm, but with limited set of global test vectors to these optimal (1-st order) 4 test vectors are required for achievement of FCmax(Logical) too (see Table 6.9). According to Algorithm 1 – Greedy, only 3 vectors are required. The dedicated function to test this block consist of 27 assembler instructions. It should be noticed that logical instructions are used in other functions too, as supplementary.

9. Shifts operations Block

There isn't any 1-st order fault in this block. Testing this block both according to Greedy Algorithm or to Hybrid Algorithm, only 2 test vectors are required for achievement of FCmax(Shift). The dedicated function to test this block consist of 55 assembler instructions. It should be noticed that logical and arithmetic instructions are used in function to SHIFTs testing too, as supplementary.

10. Arithmetical Operations Block.

One 1-st order fault is found in this block. So I applied the Hybrid Algorithm first, and this results in 4 test vectors required to FCmax(Arithmetical). Whereas Algorithm 1- Greedy requires only 3 test vectors (see Table 6.9). The dedicated function to test this block consist of 29 assembler instructions. It should be noticed that arithmetical instructions are used in other functions too, as supplementary.

11. ALU Multiplexer.

There is not direct access to the ALU Multiplexer block from instructions level. So the ALU Multiplexer is tested indirectly by execution of functions for LOGICAL, ARITHMETIC and SHIFT instructions testing and a special function for solving problems related to HW redundancies in this block has been written. Together about 300 instruction test the ALU Multiplexer in indirect way. Greedy Algorithm requires only 2 vectors for achievement of FCmax(ALUMux). Algorithm Hybrid requires 3 vectors. Results are shown in Table 6.9.

To conclude this part of considerations only 9 test vectors are required for achieving total FCmax for Logical, Arithmetic, Shift and ALU Mux blocks together. Another conclusion is that maybe a good optimization idea is to find a block which requires the higher number of 1-st order vectors. These vectors should be determined with Algorithm Hybrid. Then I can try to cover remained blocks with these vectors. When it appeared impossible or the number of test vectors was similar as for this one block which required the highest number of test vectors, then better solution would be to apply Greedy Algorithm. **Notations in Table 6.9 are following:**

Optimal set of test vectors is determined according to:

g) Algorithm 1 – Greedy
h) Algorithm 3 – Hybrid

V[y][1H] – means that first order vector [1] detects higher order faults [H] in given number of blocks y

V[H][H] – higher order vector, which detects some number of higher order faults.

The same – if global vectors are the same as local vectors for given blocks.

Table 6.9: PicoBlaze blocks with optimal sets of test vectors

#	PicoBlaze block	Vectors required to FCmax (Block)	#local vect-s	Global vectors	# Inj./Det. faults [%]	1-st o. faults
1	Fundamental Control	h) F7[1H][1H]	1	F7[1][1]	2/2[100%]	0
2	Interrupt input logic, Interrupt enable and shadow Flags	h)F7[1H][1H], (detects higher order faults)	1	F7[1][1]	4/47 [8.5%]	0
3	Decodes for the control of the program counter and CALL/RET stack	h, g) F7[1H][1H], 7E[2][3]	2	F7[1][1] 7E[2][3]	67/104 [64.4%]	1
4	The ZERO and CARRY Flags	h) (FC, FB, FA, F9, F8, F6, F7, F5, F4, F3, F2, F1,F0, EE, DE, CE, BE, AE, 9E, 8E, 6D, 5D, 4D, 3D, 2D, 1D)[1][1], 7E[2][3], 7D[3][11]	28	All 28	140/179 [78.2%]	29
5	Program Counter. Definition of a 10-bit counter can be loaded from two sources	h,g)7D[3][11], F7[1H][1H]	2	7D[3][11] F7[1H][1H]	243/320 [75.9%]	4
6	Register Bank and second operand selection	g) 04[H][H], 7E[2H][3H]	g) 2 h) 3	04[H][H, 7E[2H][3H]	155/157 [98.7%]	0
		h) 2D[1H][1H], EE[1H][1H], 5D[1H][1H]		2D[1][1], EE[1][1], 5D[1][1]		
7	Memory Storing Function	h) F7[1H][1H]	1	F7[1][1]	30/35 [85.7%]	0
8	Logical	g) 54[H][H], 0A[H][H], 3E[H][H]	g) 3 h) 4	D4, 8A, 3E	225/233 [96.6%]	0
		h) 0x72[H][H], 0E[H][H], FE[H][H], 0x78[H][H]		F2, 8E, 7E, F8		
9	Shifts	g) B3[H][H],4[H][H]	g) 2 h) 2	CB, 79	148/176 [84.1%]	0
		h) 8C[H][H], 87[H][H]		F2, F7		
10	Arithmetical	g) 0x16[H][H], 0x69[H][H], 0x77[1][1]	g) 3 h) 4	The same	150/156 [96.2%]	1
		h) 77[1][1], 2D[1H][1H], FB[H][H], BE[1H][1H]		The same		
11	ALU multiplexer	g)8E[1H][1H], 71[H][H]	g) 2 h) 3	The same	133/217 [61.3%]	0
		h)2D[1H][1H], F2[1H] [1H], 5D[1H][1H]				
12	R/W Strobes	------	2		9/52 [17.3%]	0
13	Prog.CALL/RET stack	g,h) 7D[3][11], F7[1H][1H]	2	The same	66/126 [52.4%]	6

The conclusion can be driven from the above table that for most hardware blocks two local vectors are enough to obtain FCmax. The exception is arithmetical block (3 local vectors) and flags module

which requires 28 vectors. All these vectors for flags module testing are first order. Consequently, in the case when the test run time is crucial, local rather than global vectors should be used. The drawback of local vector usage is that more input and output vectors should be transferred to / from the microprocessor, and in some cases these vectors transfer may be more problematic than increased test run time in the case of global vectors. Further optimization may be achieved by employing a hybrid method: global and local vectors usage, i.e. employing two global vectors and 26 local vectors to test only specific blocks of the microprocessor. This, however, requires further researches.

Percentage distribution of detected faults in every individual PicoBlaze block is presented in Figure 6.2. We can observe that the best results are achieved for blocks which operate directly on data (Blocks 6-10 in Table 6.9). ALU Multiplexer is tested indirectly by assembler functions and in this block exists hardware redundancies. Moreover a lot of undetected faults have its place in HW which realizes I/O operations (about 39% injected in this block). For this reasons FC is here relatively low. The most difficult one to test is the ZERO and CARRY flags generation block. Hence, there is the higher number of first order faults and vectors to detect them. Hardware architecture of this block is most complex. Many one-bit details on both operand (256×256) are required to detect all possible situations related to faults in this block. Moreover a few instructions can generate the same flags in situation when all 8 bits are taken under calculations and range of register is limited to 7, 6, 5, etc. bits. FC for PicoBlaze blocks dedicated to Program Counter is low too, because testing of these processor resources is not main task of this bijective program, as mentioned above.

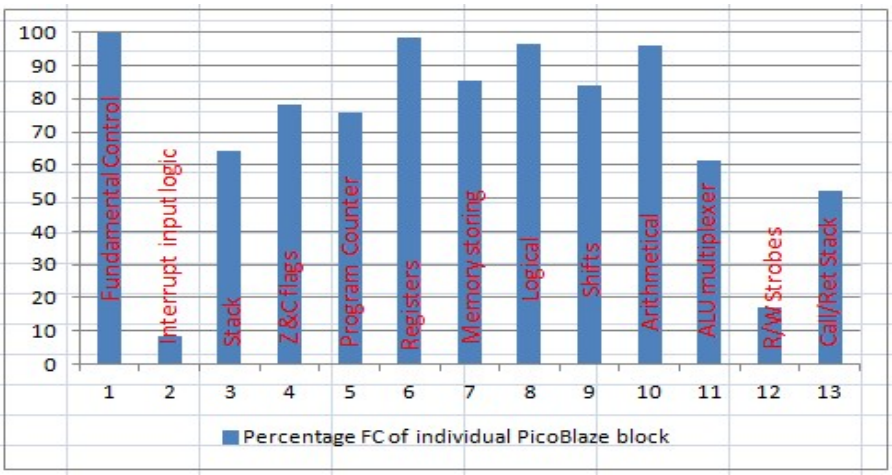


Figure 6.2: Percentage of FC of individual PicoBlaze block

6.2.5. Fault coverage in cyclic usage of results vectors

In this thesis a bijective testing is proposed. There are different types of bijective functions proposed in Section 5, e.g. identity, bit inversion or permutation, addition / subtraction a constant value, LFSR. Construction of feedback loop which covers full input vector space for some bijective function is relatively simple. For example addition by 1 may covers full input space.

Combining (serializing in the test program) bijective functions also forms a bijective function, which is used in the test program. Nevertheless construction of feedback loop which covers full input space for a combination of bijective functions is not an obvious task. Therefore I took an attempt of solving the problem of lack of the full cycle as it is further described.

The research work, presented in this subsection, has shown that one of the fastest method to achieve the FCmax is determining the optimal start vector and then cyclic generate test vectors in the loop instead of loading test vectors from external memory I took up as well, an attempt of testing of the microprocessor in cyclic way. This time principle was as following:

Algorithm 4: Cyclic usage of result vectors

Determine set F of all faults f_i
Determine the first test vector v_i *which detects the highest number of faults / or which detects the higher number of faults of first order (see definition 6.1);*
Remove all faults f_j *that are covered by test vector* v_i *from set F;*
While *F is not empty*
{ *Take the output vector and reuse it as a new input* v_i*;*
Test the microprocessor with input vector v_i*;*
Remove all faults f_j *that are covered by test vector* v_i *from set F;*
}

My practical experiments showed, that decisively better FC was achieved when the testing program is not only bijective, but it is able to generate all the possible test vectors in a cyclic way.

One of goals of this method is to spare resources. Repeated executions of the test sequence require minimal additional memory resources since only the initial test vector, the final test result, and number

of iteration are stored. If a minimum test time is required, a minimum numbers of iterations achieving FCmax should be selected and stored in the memory together with the results (or signature).

Initially the target of my observation was to learn, how many iterations were required to obtain the FCmax. It turned out that results of the bijective program which does not generate a full data cycle often are not able to achieve FCmax. There are a few vectors, which do not generate full cycle, when are used as initial vectors (0x00, 0x01, 0x0B, 0x3E, 0x77, 7F). Other initial vectors generates full cycle, and achieves FCmax. I have come to the conclusion, that good idea was to start from vector 03 or FB. For these vectors the number of iterations, which are required to obtain the FCmax equals 235. On average I need 245, iterations to obtain the FCmax.

To conclude, Algorithm 2 and Algorithm 3 – (improved lowest-order first) bring usually better results than Algorithm 1. The experiment 6.1.2 showed, that if all vectors of first order were executed (28), 100% of FCmax was achieve We can observe in Table 6.2, that all the vectors of first-order must be applied in order to achieve the FCmax. And it may happened that vectors of higher order will needed too. Algorithm 4 gives the worst results, while it is the simplest and can be easily used in industrial production testing.

6.2.6. Number of iteration at Algorithm 4

As described in chapters 6.2.5, the test sequence can be executed with input test vectors used in a cyclic way. The result of the previous execution of the test sequence can serve as the new input test vector of the next iteration of the test sequence. The results of such methods are shown in Figure 6.3. After the first vector has been tested different FC are obtained: the initial input test vector 7D (first-order vector which covers the highest number of first-order faults) gives 78.33% FCmax. Vector F7 increases the fault coverage to 90.63% FCmax. In the next step, with the resulting vector 83, the fault coverage increases to 95.1% FCmax, etc. In this particular case 8 runs of the test sequence were required to reach the 97% FCmax. Another example is which the starting vector is the first order vector (F2) which detects the highest sum of faults too (in total 90,23%). In this case 33 iterations are required to obtain 97% FCmax. Whereas when I started with the best vector B1 (the highest total FC equal 90.7% FCmax), resulting test vector 98 gives aggregated FC 93.33%, in the third iteration resulting test vector CD increases FC to 95.15% and so on. To achieve 97% of FCmax, 21 iterations are required in this case. Result of the experiment are presented in Figure 6.3 and Table 6.10.

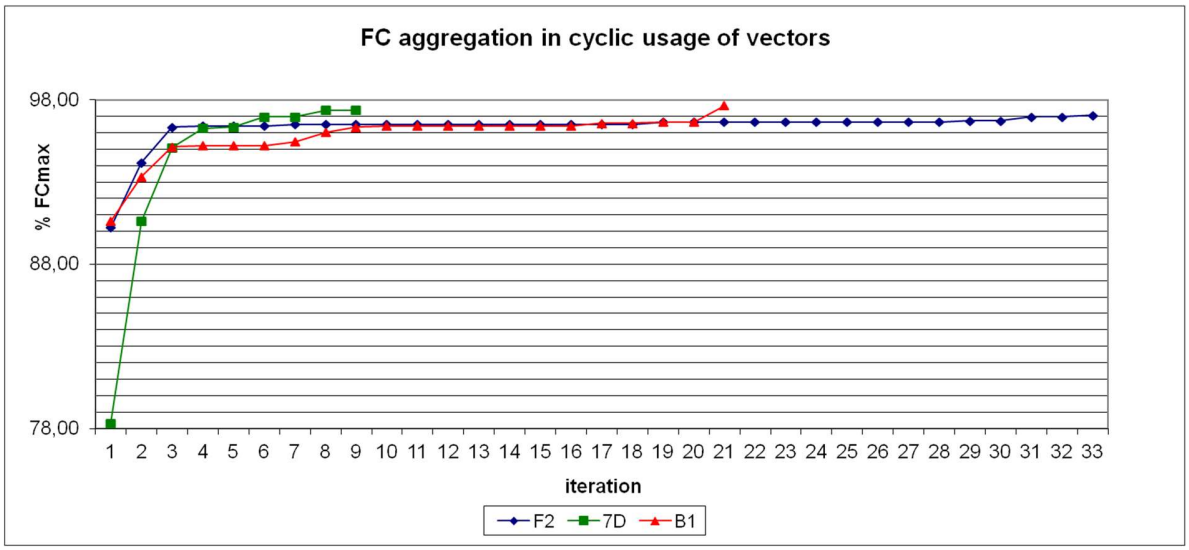


Figure 6.3: Achieved fault coverage during cyclic usage of test vectors

Table 6.10: FC aggregation in cyclic usage of vectors

Iteration	Starting vector 1-st order **F2**		Starting vector 1-st order **7D**		Starting vector the best **B1**	
--------	vector	% FCmax	vector	% FCmax	vector	% FCmax
1	F2	90,23	7D	78,32	B1	90,63
2	99	94,20	7F	90,63	98	93,33
3	47	96,35	83	95,1	CD	95,15
4	72	96,43	7B	96,27	E0	95,23
5	19	96,43	8B	96,34	BD	95,23
6	C7	96,43	6B	96,98	80	95,23
7	F4	96,51	AB	96,98	FD	95,47
8	95	96,51	AC	97,38	FF	96,03
9	4F	96,51	24	97,38	03	96,35
10	62	96,51			FB	96,43
11	39	96,51			0B	96,43
12	06	96,51			EB	96,43
13	71	96,51			2B	96,43
14	97	96,51			2A	96,43
15	53	96,51			AA	96,43
16	5A	96,51			28	96,43
17	CA	96,51			AE	96,58
18	68	96,51			20	96,58
19	2D	96,66			BE	96,66
20	1E	96,66			0	96,66
21	41	96,66			7D	97,67
22	76	96,66				
23	11	96,66				
24	D7	96,66				
25	D4	96,66				
26	54	96,66				
27	D6	96,66				
28	50	96,66				
29	DE	96,74				
30	40	96,74				
31	7E	96,98				
32	01	96,98				
33	F7	97,06				

6.3. Problem of non-full cycle of results

Often at industrial testing or testing of dedicated FPGA processor core it may turn out, that an approach consisting in test pattern generation in the loop executed on the tested core may be inadvisable and ineffective. Such a loop is built on the basis of assembler instructions as add, jump, load etc. Worth of notice is that instruction needed to construct such a loop are different sorts (arithmetical, jump, logical). So it is obvious, that it may generate wrong unwitting input vectors as a result of injected faults or SEU induced faults too. Moreover a lack of generation of input test vector may occur. This can narrow the scope of testing vectors. Here a situation of not detected fault may appear at lack of test vector, which detects the fault. In such a case it would be hard to say, that a processor was tested in exhaustive way. Thus probability that a fault can affect this loop is relatively high. The same problem exist in case of test pattern generation with LFSR composed from a processor assembler instructions. For these reasons one of the most important industrial tests before production consists in direct introduction of test vectors into tested circuit from reference memory, and storing back results to the memory. Significant companies on the marked such as STMicroelectronics in Agrate (MB, Lombardy, Italy) and Catania (CT, Sicily, Italy) apply this test.

The method from chapter 6.2.5 leads to obtaining the FCmax exclusively, if I have such set of reference results (i.e. obtained after execution of the program without fault injection) that each result after execution program denotes different unique, next input vector. So, input vectors do not repeat.

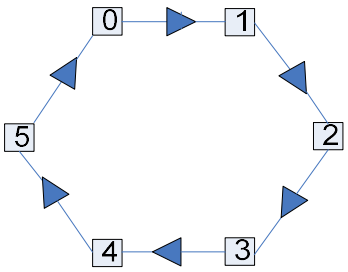


Figure 6.4.a: One full cycle

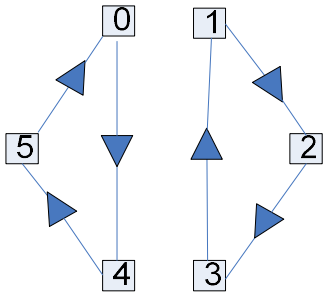


Figure 6.4.b: Two cycles

For example let us consider the number of only 6 test vectors values: 0, 1, 2, 3, 4, 5. In the first case I have cyclic group of inputs vectors determined by all output results. This means every input vector from among the whole the set is unique. In the second case I have not one cyclic group of input vectors because the result vector determines again vector „0" as input. The example of the full cycle is presented in Figure 6.4.a, and the example of the two cycles is presented in Figure 6.4.b.

According algebraic theory of group **(Wikipedia 2020 B)**, a cyclic group is a group that can be generated by a single element, in the sense that the group has an element *g* called a „generator" of the group such that, an element *gj* composed with itself generates all other elements of cyclic group $gi = gj \circ gj$. The composition in my case is the test program executed by the microprocessor.

The problem of non-full cycle was observed during writing my test program. For example in the previous version of the program the trial of obtainment fault coverage in cyclic usage of results was failed. There are available 256 generators of cyclic group, which are complete set of possible test vectors for the PicoBlaze microprocessor. Generation of the cyclic group with each of these generators led to obtain a subgroup in case of the every vector.

When bijective blocks of the program were analyzed by a script-program in order to check if generated by them outcomes form cyclic groups, it turned out that for lack of full cycle responsible were bijective blocks to testing shift instructions. The operation of shifts alone makes impossible generation of all possible numbers from the range of 0 to 255.

Moreover construction of bijective blocks, were carry from LSB or MSB after „SHIFTS" is captured, rotated and combined into data flow turned out as not enough too. Such a solution narrows the scope of data and cause that the data for testing next blocks is not exhaustive, what considerably worsens fault coverage. Experiments with changes of order of bijective blocks did not bring the expected results. The number (8) of such a block to test shifts is quite high. So event in case of program composed exclusively from block to testing shift instructions, results from one block are not enough to exhaustive test the next block (Lack of data). Similar situation appears when I composed more than one block to testing shifts together with others block to testing remaining instructions. It is known, that composition of some block of program from two bijective instructions in feedback does not necessarily generate a fully cyclic results. It works in case of more than two bijective instructions or block too.

6.3.1. LFSR – as a solution of the non-full cycle problem

My investigation of non-full cycle problem is application of Linear Feedback Shift Register (LFSR) to generation exhaustive seed of test vectors for all shift and rotate instructions of PicoBlaze processor. I led trials of cyclic testing of all among the shift instruction in their penultimate version of bijective blocks. For reminder, every of such block tested separately generates non full cycle of new input test vectors. The bijective block to testing of cyclic property of individual shift instruction is presented in Figure 6.5.

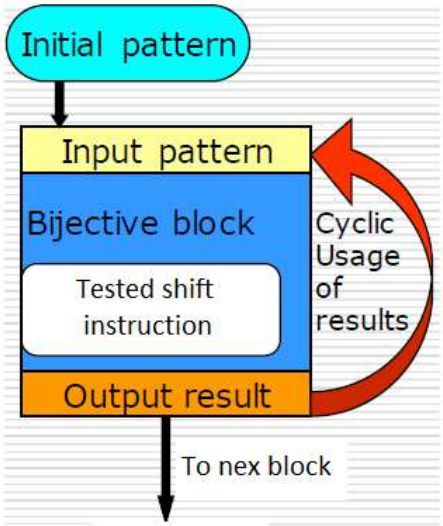


Figure 6.5: Bijective block for testing of cyclic property of individual shift instruction

Testing cycle of such a block is on average 64 vectors long, and how fast a cycle is completed depends on starting test vector and architecture of a bijective block. Each among these blocks consists of additional instructions to input carry generation, to combining of the bit shifted to carry by the tested shift instruction, etc. So constructed according to the above principles bijective blocks, every for individual shift instruction testing, have limited performances of generation of exhaustive set of test vectors 255 (seed). It leads to serious difficulties in constructing of fully cyclic bijective testing program composed of such bijective sub-blocks with lack of full cycle property.

This inconvenience can be overcome by construction of bijective and able to full cycle generation blocks to target shift instructions testing, built on base of Linear Feedback Shift Register (LFSR). Such a LFSR possess its characteristic polynomial **(Hlawiczka 1997)**. In these new blocks key role plays

a target shift instruction with additional XOR and TEST instructions which are utilized to test bits pointed by the characteristic polynomial of the LFSR **(Wegrzyn 2014 B).**

Such the polynomial determines unique configuration which performs generation of exactly determined number of resulting test vectors. Here are known architectures of Fibonacci and Galois. In my case study, it can be easily applicable Fibonacci architecture of LFSR depicted in Figure 6.6. Characteristic polynomial to generation of 255 different test vectors (for 8-bit shifted register) by cyclic usage of results of execution of every shift instruction has form:

$$X8 + X6 + X5 + X4 + 1$$

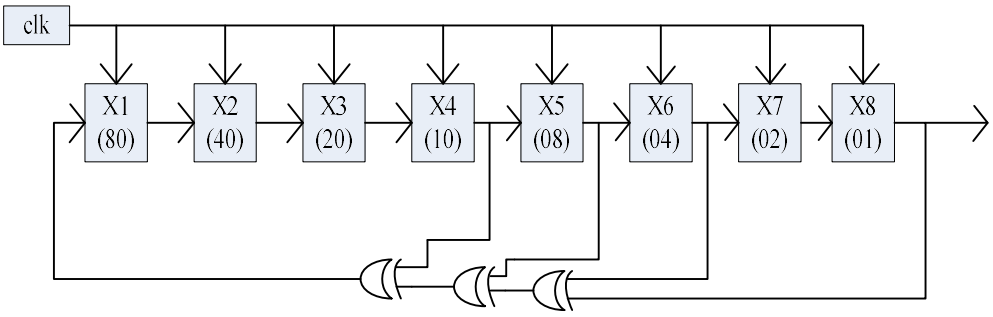


Figure 6.6: Circuit representation of 8-bit Fibonacci LFSR

My program is constructed in this way, that for every shift instruction a block consisting of additional „TEST" instructions and „XOR" instructions is developed. In some cases, when need occurs, additional „AND", „RR", „RL", „OR" instructions are utilized to clear or set, for instance, certain bits, which are set to the same values default at execution of certain shift instructions.

Moreover all the „right" SHIFT instructions are tested in one common cyclic block consisting of cyclic sub-blocks. Every such sub-block is developed to testing of individual shift instruction. Analogically all the „left" shifts or „ROTATE left", „ROTATE right" instructions are tested in other adequate common cyclic blocks consisted of cyclic sub-blocks. The main program architecture is presented in Figure 6.7, and the example of the assembler program block to the Shift Left „1" fill (SL1) test is presented in the Listing 6.1.

Every such common block is tested by cyclic generation of 254 unique output results. Every time the previous output result serves as the next input vector.

In this way after the block of program is executed 254 times, one unique (255-th) test vector is generated. This test vector is different for every unique input test vector. Then the 255-th result vector from the cyclic „right shifts" block serves as input test vector for „rotate" cyclic block of instructions. Next cyclic block is again tested by cyclic generation of 255 unique output results, and so on. Of course the starting test vector can be whichever from the range 1-255. And according to the main principle of seed generation with LFSR with full cycle polynomial after 254 cycles every result corresponds one to one with its input argument vector. This test vector serves as input vector for a next block of program.

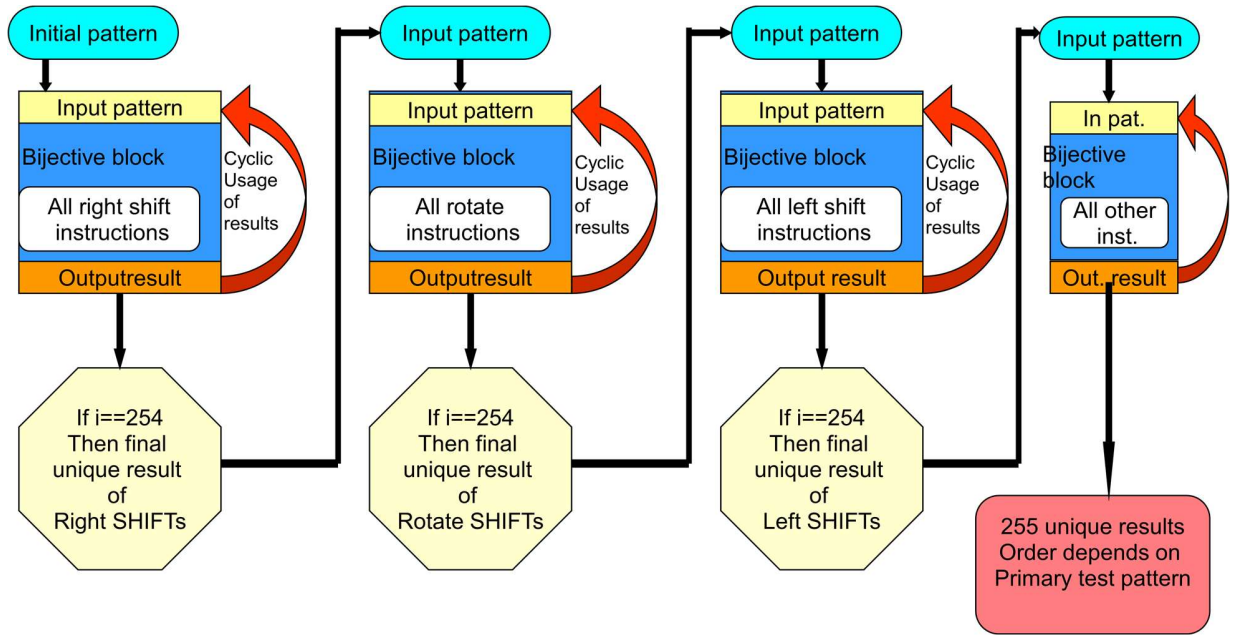


Figure 6.7: The program architecture

```
next1:
        LOAD    sC,   s0;
iterac1:                ;    by these two instructions  I
        TEST    sC,   80;    can access intended bit in
        ADDC    s9,   00;    the register. In this way inputs
        LOAD    s8,   s9;    are prepared for ,,XOR" LOAD s8, s9
        LOAD    s9,   00;    All these code is adequate to circuit in
        TEST    sC,   20;
        ADDC    s6,   00;
        XOR     s8,   s6;
        LOAD    s6,   00;
        TEST    sC,   10;
        ADDC    s5,   00;
        XOR     s8,   s5;
        LOAD    s5,   00;
        TEST    sC,   08;
        ADDC    s4,   00;    generating a bit
        XOR     s8,   s4;    to substitute instead
        LOAD    s4,   00;    bit ,,0" of shift
        TEST    s8,   01;    bit ,,0" of shift SL1
        SL1     sC;
        AND     sC,   FE;
        OR      sC,   s8;
```

Listing 6.1: Example of a bijective cyclic block of test program to the Shift Left ,,1" fill (SL1) test

Corresponding circuit:

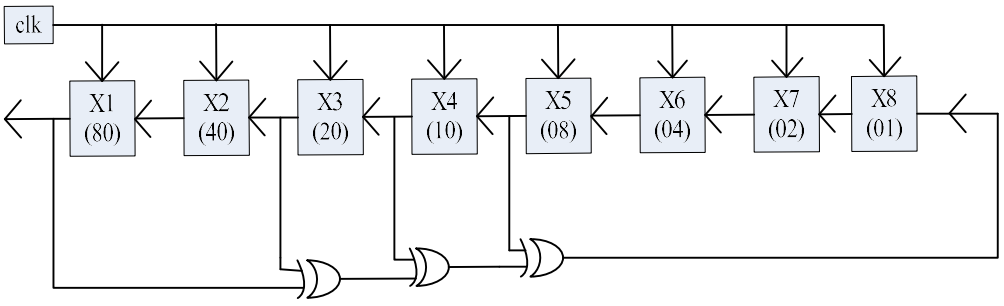


Figure 6.8: Bijective cyclic block to the SL1 testing

This solution has brought effect especially for the SHIFT blocks testing. Almost all the faults have been detected here. Application of the solution of LFSR caused rise of detect-ability in such a block as for arithmetical and logical instruction testing. Here is possible to observe two roles of LFSR based block to shifts testing:

- self-test to shift instruction testing,
- exhaustive seed generator for other blocks of test program.

6.3.2. Results of refinement of the PicoBlaze test program using LFSR

The final version of the test program with built-in LFSR to testing each of the eight PicoBlaze SHIFT instructions, and two rotate instruction achieved fault coverage of 94,76%. This program generates all input carry bits, which are captured by these SHIFTs too. In first stage of the experiment, I had applied 255 vectors as the algorithm assumes, and then I have discovered, that this had been redundant because here individual vectors exist, which can detect alone about 94,76% of injected faults (1519/1603) using exclusively one test vector. This is the main advantage of this refinement. This 94,76% is called FCmax gained by my test program. Furthermore there exist 106/255 of such vectors. This is 41% of total number of 255 vectors. Every among next 61 vectors can detect almost all faults without 1 fault (1518), 24 vectors can detect less by 2 faults (1517), 15vectors detect 1516 faults, 6 vectors detect 1515 faults, 9 vectors detect 1514, and so on (see table 6.3.11). The weakest vector detected 1102 faults. Described here selected cases of testing vectors with their Fault Coverage in number of detected faults are collected in Table 6.3.12.

Table 6.3.11: Number of faults detected by percentage of test vectors

Detected faults	1519	1518	1517	1516	1515	1514	1513	1512	1509	1508	1507
% total vectors	41,4	23,8	9,4	5,9	2,3	3,5	1,2	0,8	0,8	2	1,2
Detected faults	1504	1503	1501	1500	1499	1498	1495	1491	1497	1465	1102
% total vectors	0,8	0,4	0,4	1,6	1,2	1,2	1,2	0,4	0,4	0,4	0,4

Table 6.3.12: Selected cases of the Fault Coverage for testing vector

Vectors achieve full FC 1519 f. detected								Vectors 1518 f. detected											
Nr	Vector		Nr	Vector		Nr	Vector	Nr	Vector		Nr	Vector		Nr	Vector				
1	82		42	28		83	AE	1	2B		43	62		21	6F				
2	83		43	29		84	B1	2	AC		44	E8		22	DC				
3	84		44	AB		85	B6	3	AF		45	05		23	9B				
4	85		45	AD		86	38	4	B0		46	8E		24	66				
5	86		46	2E		87	BC	5	B2		47	91							
6	06		47	B3		88	C2	6	B4		48	17							
7	87		48	34		89	44	7	3B		49	A5							
8	88		49	B5		90	49	8	BF		50	31							
9	08		50	B7		91	4A	10	41		51	42							
10	89		51	B8		92	D1	11	45		52	D3							
11	8A		52	B9		93	D5	12	C7		53	5C							
12	0A		53	BA		94	E9	13	4C		54	64							
13	8B		54	BB		95	EE	14	4F		55	0F							
14	8C		55	BD		96	1C	15	50		56	2C							
15	0C		56	3E		97	22	16	51		57	4B							
16	8D		57	3F		98	A9	17	53		58	ED							
17	0E		58	C1		99	37	18	57		59	36		The weakest vec.					
18	90		59	C3		100	BE	19	58		60	A2		Nr	vector	FC			
19	11		60	43		101	C8	20	5B		61	D8		1	FB	1507			
20	92		61	C4		102	52	21	DD		1517 f. det.			2	F8	1507			
21	12		62	C5		103	33	22	5E		Nr	vector		3	7B	1507			
22	94		63	46		104	3A	23	E4		1	DA		1	7E	1504			
23	14		64	47		105	4E	24	E5		2	DE		2	FA	1504			
24	95		65	C9		106	56	25	EA		3	E0		1	78	1503			
25	96		66	CA				26	EF		4	E3		1	7C	1501			
26	16		67	CB				27	07		5	E6		1	F4	1500			
27	97		68	CC				28	8F		6	E7		2	FF	1500			
28	98		69	CE				29	10		7	EC		3	F6	1500			
29	18		70	CF				30	93		8	09		4	F3	1500			
30	99		71	D2				31	1E		9	13		1	72	1499			
31	9A		72	D4				32	A1		10	19		2	F2	1499			
32	1B		73	54				33	23		11	21		3	F7	1499			
33	9C		74	D6				34	A8		12	30		1	75	1498			
34	9D		75	68				35	2F		13	D9		2	73	1498			
35	9E		76	EB				36	32		14	DF		3	76	1498			
36	1F		77	6C				37	C0		15	63		1	71	1495			
37	A0		78	F1				38	C6		16	67		2	F5	1495			
38	A3		79	9F				39	CD		17	15		1	F1	1491			
39	A4		80	24				40	D0		18	AA		1	70	1479			
40	A6		81	27				41	D7		19	39		1	00	1465			
41	26		82	2A				42	5A		20	E2		1	77	1102			

The finally achieved fault coverage for the proposed approach LFSR supported has been 84.2%. If I neglect the faults related to the input / output operations I have got 94.76% of fault coverage.

Investigation of fault coverage concerning input/output buffers is unprofitable due to simulation time. There is 256 simple I/O buffers. Hence I should emulate 256 x 256 input vectors to test them. There are no peculiarities nor redundancies in the construction of these buffers, thus I can expect high fault coverage for these blocks. So testing of I/O buffers wasn't object of my work. Usually testing of I/O buffers is matter of separate experiments **(Chen 2001)**, which are not BIST based. Exist advanced solutions of test intended to I/O blocks in the bibliography of the subject **(Stroud 2009), (Haar 2007), (Pereira 2005)**. Final results are presented in Table 6.3.13 below:

Table 6.3.13: Fault coverage. Final results

Test program	**FC (%) Stuck-at faults**		**FC (%) Complete list**	
	934 faults	883 faults	1804 faults	1603 faults
fibonacci	49.9	52.8	33.8	37.7
fibonacci (recursion)	56.0	59.2	43.3	48.4
random instruction order	62.7	66.4	46.8	52.3
MM1	63.6	67.3	49.9	56.1
MM2	64.4	68.1	51.2	57.6
MM3	66.4	70.2	53.1	59.7
data sensitive path (not completely bijective)	72.6	76.8	62.9	70.2
data sensitive path + bijective sub-sequences	88.1	93.2	76.6	85.6
data sensitive path + bijective sub-sequences + LFSR + additional manipulations	90.3	95.4	84.2	94.76

An analysis of the remaining 5,24% of the faults that were not detected showed that 1.2% corresponded to specific fault situations – their detection requires specific values loaded in given registers plus specific values of status flags (i.e. zero result, carry) as a result of the execution of previous instructions. Such faults are difficult to detect with the automatic generation of functional tests. In this case simply interchanging the order of the sub-sequences turned out usually to be insufficient. Here slight improvement of fault coverage was achieved by a manual modification of the test sequence. And also these faults are not detectable regardless of the any test program, due

to a few kinds of logical or hardware redundancies as I have analyzed in chapter 9 about fault masking. A total of 4% of the faults proved to be due to the Program Counter locations, Decoder for control PC, CALL/RETURN stack, and CALL/RETURN stack. My testing program occupies only about 370 from 1024 available PC locations. Experiments with the copying of the program to other locations yielded results. However, it is not profitable to perform a fault simulation investigations, with a test program copied three times to all remaining free PC locations, because the time of such researches increases proportionally. In the bibliography of subject exist many exhaustive solutions of tests dedicated to testing PC and stack. Such solutions often are hardware based, or utilizes both hardware and software designed for test **(Campbell 2014)**, **(Sanchez 2011)**. There still remain 14 undetected faults inside „SHIFTs" block, 4 in the arithmetical instructions block, and one fault in the logical instructions block. Further, in chapter 9 detailed analysis of these faults are carried out. Regarding neglected faults, 84 undetected faults are related to the Input Port connected to „ALU" multiplexer, 43 to Read/Write Strobe, and 38 to Interrupt inputs and Interrupt logic.

The next important issue is time of the complete test sequence execution. Initially the maximal fault coverage had achieved by usage only 28 test vectors for earlier version of bijective, but not fully cyclic program without LFSR support. In this case time of execution of these 370 instructions 28 times is estimated at 400µs. Time of one execution of 370 PicoBlaze assembler instructions can be estimated at several dozen of microseconds. For comparison, testing of the PicoBlaze with a method of read back of its configuration could take about 3,5 s at the same equipment. My latest newest researches have revealed, it is enough to apply exclusively one test vector to achieve the full fault coverage, if large percent of the sensitivity paths is activated. This has been assured by the last version of fully bijective test program supported by LFSR solutions. Moreover I have found more such test vectors which achieved 100% of achieved before fault coverage at usage of exclusively one vector.

6.3.3. Comparison of results

Table 6.3.14 presents a summary of the researches results from bibliography of the subject **(Psarakis)**. These research results on the processors whose functionality, construction complexity or performance can be compared with the PicroBlaze were selected and collected in Table 6.3.14 to compare with the results achieved by my program intended to PicoBlaze testing. Results of these researches are expressed usually as coverage of injected faults into hardware of the given microprocessor / microcontroller.

The most important novelty introduced hereby is a different model of injected faults. This model differs significantly from the conventional stack-at models widely used for testing processors / microcontrollers implemented on ASIC / embedded platforms because a SEU-induced fault affects the logic elements implemented by the look-up tables (LUTs) in this manner, that the logic function is arbitrarily changed as described in details in the chapter 8.3 about fault injections.

Table 6.3.14: Summary of results of researches from bibliography in comparison with PicoBlaze

Nr	Author	Tested Processor	Year	Fault coverage [%]
1	Lingappan and Jha	Parwan	2007	96
2	**Wegrzyn**	**PicoBlaze**	**2014**	**95,4 stuck-at**
3	Bernardi, Sonza	Intel 8051	2004	95 (stuck at 0,1)
4	Zhang	Parwan	2013	94,8
5	**Wegrzyn**	**PicoBlaze**	**2014**	**94,76**
6	Krstic et al.	Parwan 8-bit	2002	92
7	Shen and Abraham	GL85 (8085)	1998	90
8	Corno, Sonza	Intel 8051	2002	89 RTL description
9	Corno, Sonza	Intel 8051	2002	85 gate level

It is worth to notice that authors of publications compared in Table 6.3.14 mainly apply stuck-at fault models, while I have injected both stuck-at, and ,,SEUs in LUTs" modelling faults. Despite the fact that faults induced by SEU in LUTs are harder to detect, I have obtained results comparable to other authors of publications, which utilized only stuck – at fault model.

7. MicroBlaze case study

As initial case study for my experiments MicroBlaze processor core was chosen. The first idea was implementation of bijective test program composed from all assembler instructions of this processor. The most important principle was, that all the data should be preserved, and all sensitive paths should be activated, similarly as in case of the PicoBlaze test program described in chapter 5. I had written initial version of the MicroBlaze test program, and then I interrupted further development of this program because problems with evaluation of its efficiency appeared. I have described these problems in the next chapter. As a new case study I have chosen PicoBlaze. Then, because both these processor cores were dedicated those days for the same FPGA families, I decided utilize experience gathered at PicoBlaze researches and to study possibilities of implementation of the MicroBlaze bijective program accordance to principles developed for PicoBlaze. Results of my investigations are presented in chapters 7.5 and 7.6.

The MicroBlaze embedded soft-core is a pipelined reduced instruction set computer (RISC). The register set consists of thirty-two 32-bit general-purpose registers, a 32-bit address bus, a Program Counter (PC), a Machine Status Register (MSR), an Exception Address Register (EAR), and an Exception Status Register (ESR). The core includes a single-issue pipeline, a data and instruction cache, and it supports Fast Simplex Link (FSL) or in latest version AXI Stream for coprocessor/peripheral interface. The hardware multiplier is (optionally) implemented in Virtex-II and subsequent devices. All the MicroBlaze instructions are 32 bits or 64-bit for 32-bit immediate arguments. The instructions can be categorised as follows: arithmetic, logical, shifts, branch, load/store and special. MicroBlaze uses a pipelined instruction execution. The pipeline was divided into three stages: fetch, decode and execute, in the case of earlier versions. Currently number of the pipeline stages are configurable (8 stages max). For most instructions, each stage takes one clock cycle to complete. Consequently, it takes at least three clock cycles latency for a specific instruction to complete in case of three-stage pipeline. The instruction set consists of roughly 110 instructions. In the instruction test they are combined into an information-sensitive path in such a way that the result reflects the composite contributions of the outputs of the individual instructions executed along the data corruption (fault injection) path. In the other words, the test program should be arranged in such a way that the information written in individual bits does not disappear, but affects the final results.

7.1. Description of selected instructions

According the MicroBlaze user guide this processor uses two instruction formats: Type A and type B. Type A instructions have up to two source register operands and one destination register operand. Type B instructions have one source register and a 16-bit (or 32 bit) immediate operand. These instructions have a single destination register operand. Type A is used for register-register instructions. It contains the opcode, one destination, and two source registers. The bit map of the type A instruction is presented in Figure 7.1. Type B is used for register-immediate instructions. It contains the opcode, one destination, one source register, and one source 16-bit immediate value. Type B instruction is respectively in Figure 7.2.

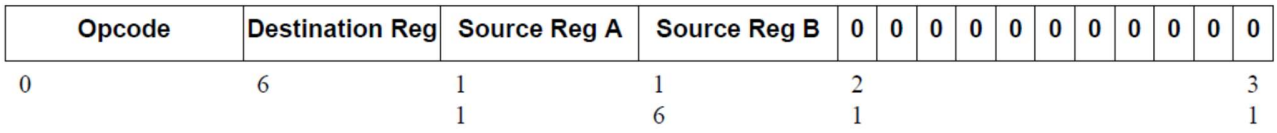


Figure 7.1: The Bit map of the type A MicroBlaze instruction

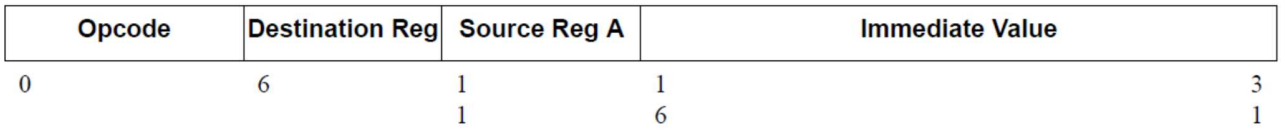


Figure 7.2: The Bit map of the type B MicroBlaze instruction

In the test program MSR (Machine Status Register) is used. This register content keeps control and status flags, e.g. the carry flag MSR[29]. Shortened description of some instructions used in the test program (MicroBlaze reference guide) is as following (see also attached CD):

Arithmetic ADD:

There exists four functional varieties of the instruction:

Add rD, rA, rB; ADD
Addc rD, rA, rB; Add with Carry
Addk rD, rA, rB; Add and Keep Carry
Addkc rD, rA, rB; Add with Carry and Keep Carry

The sum of the contents of registers rA and rB, is placed into register rD. Bit 4th of the instruction labeled as C (CARRY) in the Machine Status Register (MSR) is set to one for the mnemonic addc. Both bits are set to one for the mnemonic addkc. When bit 4 of the instruction is set to one (addc, addkc), the content of the carry flag (MSR[C]) affects the execution of the instruction. When bit 4 is cleared (add, addk), the content of the carry flag does not affect the execution of the instruction (providing a normal addition).

Logical AND :

And rD, rA, rB;

The content of register rA is ANDed with the content of register rB. The result is placed into register rD.

Barrel Shift Right Logical Immediate BSRLI:

Bsrli rD, rA, IMM;

Shift the contents of register rA by IMM bits and puts the result in register rD.

Move From Special Purpose Register:

Mfs rD, rS;

Copies the contents of the special purpose register rS into register rD.

Store word:

Sw rD, rA, rB;

Stores the contents of register rD, into the word aligned memory location that results from adding the contents of registers rA and rB.

Load Word:

Lw rD, rA, rB;

Loads a word (32 bits) from the word aligned memory location that results from adding the contents of registers rA and rB. The data is placed in register rD.

7.2. Initial MicroBlaze test program

The first example **(Wegrzyn 2007)** shows how the result of an arithmetic instruction (i.e. the sum of the contents of two registers) and the status flags are combined. The execution of some instructions affects the status flags (the zero flag, carry, etc). In order to detect possible faults in the status information, the contents of the status register are included in the result of the currently executed instruction. This is normally achieved by „XOR-ing" the contents of the status register and the resulting output data or captured by other instructions, which functionality performs it. In the case of „ADD" instruction usually carry flag is combined with the result. The resulting output serves as the input argument for the next instruction, etc. similarly like it is for the PicoBlaze. A few instructions require multiple clock cycles to complete. Listing 7.1 of the test program **(Wegrzyn 2007 B)** checks the correct operation of instructions „ADD", „ADDC", „MFS" on registers R2, R3, R4.

```
add   R2, R3, R4;    // R2 = R3 + R4
mfs   R3, Rmsr;      // copy status register Rmsr to R3
addc  R2, R2, R3;  //R2= R2 +R3  + carry
```

Listing 7.1: ADD and MFS instructions testing

In the Listing 7.2, block of code bellow BSRLI (Barrel Shift Right Instruction by immediate value (2)) is primarily tested. Shift instructions can lose information about n-LSB or n-MSB, for n-bit right or left shits, respectively. In order to prevent this loss the XOR operation with the original contents (before the shifting) is performed to regenerate the information.

```
and   R5, R3, R4;  //the outcome of the previous instruction is stored in R3, R4
bsrli  R3, R5, 2; //shift logically right by the amount specified by the immediate value (2)
xor   R2, R3, R5;  //combine the outcome before shifts and after shifts by XOR
```

Listing 7.2: BSRLI instruction testing

In some cases, current register contents are stored into a memory in order to prevent information loss which can occurs when the next instruction is executed on the same register. For example in the Listing 7.3 Store Word (sw) and Load Word (lw) instructions are tested. Contents of register R2 is stored into memory location which results from adding the contents of registers R6 and R7, and then loaded back, but to the register R3 from the same address.

```
sw    R2, R6, R7;    // load the result (stored in R2) into memory. Address = R6 +R7
and   R2, R2, R8;    // R2 = R2 and R8
lw    R3, R6, R7;    // load the previous result to R3 from address = R6 +R7
add   R2, R2, R3;    // combine the outcome of „AND" operation with the previous
                     // result and store into R2
```

Listing 7.3: SW and LW instructions testing

In the above program individual assembler instructions and blocks of instructions are bonded by „XOR" instruction. The attempt of composition of the whole bijective program with architecture of one code block has been taken up in this way. Assurance bijective property of this program by „XORing" data and status flags turned out to be a difficult task. There were too many overlapping sub-problems, related to all the status flags, results with fixed values reminded still on the same bits, branch instruction execution etc.

7.3. Evaluation of the MicroBlaze fault coverage

In order to evaluate the fault coverage of the processor-core test some means of faults injecting needs to be made available **(Wegrzyn 2007)**. Since the HDL source code of the MicroBlaze embedded soft core is not available to ordinary users, software fault-injection into the processor core in similar way as for PicoBlaze is not feasible. Speculations about breaking the CRC protection of the configuration file and changing the values of individual bits could lead to serious damage of the FPGA circuit, and they are therefore not recommended.

These experiments were carried out using the Spartan3E board with the MicroBlaze implemented on this board. MicroBlaze test program was executed on the Spartan 3E board operating at main clock speed of 50 MHz. Input test vectors were generated on PC and sent to Spartan 3E GPIO A from PC serial port to ensure controllability. Output results were transmitted back from Spartan 3E GPIO B to PC via parallel PC port. Table 7.1 presents number of generated test vectors and percent coverage of full vector range.

Table 7.1: Time of experiments vs percentage coverage of the test

Number of applied test vectors	Simulation time [h]	Coverage of full vector range [%]	Starting vector
55287360	64,8	1,29	0x00000000
58129920	68,1	1,35	0x000FFFFF
70139520	82,2	1,633	0x7FFFFFFF

Fault injection was realized by additional „XOR" instructions inserted experimentally in a few different positions in the test program. This „XOR" alters the intermittent result according to the principle as stated in Figure 7.3. Registers R3 and R4 before execution of Instruction one are „XORed" with mask registers RXor and RXor1 respectively. Before execution of Instruction 2, register R2 is „XORed" with the mask register RXor 2. The correct logical values can be disturbed at least once in each bit of the intermittent results in this way. The maximum reasonable number of „XOR" instructions inserted is 39. The entire test program consists of approximately 110 instructions. It should be noted that only about 70% of these instructions directly concerns the data. The remaining 30% are instructions such as „Branches", instructions for reading state registers, etc.

According to this methodology, I provided the possibility of injecting 39 faults simultaneously or every fault one by one. When I injected all the possible 39 faults simultaneously, achieved fault coverage was 100% as expected. In the first experiment I proceed in such a way, that faults are injected into places where I expect appearance of data, in accordance with the selected start vector. I inserted 39 „XOR mask" after / before instructions which perform operations on data. First, mask is selected in order of bit corruption and a chosen bit position from the range from 0 to 31 is affected. Next testing vectors are generated in loop: current_vector + 1. Obviously, it is not effective method of generation input patterns for 32-bit processor. Generation of a test pattern with LFSR and then signature analyzes should be applied instead. However due to impossibility of evaluation of the test program by hardware fault injection, signature analyzes were not implemented. This experiment has shown how long time is required to obtain on average about 1,5% of results, where the complete set of results is 2^{32} output vectors. Finally I observe if this fault was detected using C script. While faults were injected one by one, detect-ability depended strongly from position of injected faults and chosen input data. The results are presented in Table 7.2.

For comparison an another experiment was carried out, where the fault positions of injected faults were drown randomly using C functions rand(). As before, in the first random experiment I inserted 39 „XOR mask" after / before instructions which perform operations on data. First mask in order of inserting is selected an a bit position from the range from 0 to 31 is drawn randomly. Exclusively on this bit position a fault was injected by the „XOR mask". Then the first, second and third series of test vectors were executed, and finally I observed using C script if this fault was detected. Next the procedure of randomly fault injection is repeated for all reminded „XOR masks" one by one, once for the first argument register (in summary 39 faults), then for the second argument register (39 faults), and eventually for both these registers (in summary 2 x 39 faults). In this case two faults were injected concurrently in the test program. The results are presented in Table 7.3 below.

In the second random experiment I inserted 39 „XOR mask" identically as in the first experiment. This experiment differs from the first one in a number of injected faults. Faults were drawn randomly without repetitions five times. So complete number of faults were 195 for every register, and 390 in the next experiment were faults were injected in both argument register one fault. This gives two faults concurrently. The results of second random experiment are presented in Table 7.4.

These experiments should be treated only as an overview or illustrative, due to the relatively very small number of test vectors used and the extremely long experiment time. Here were applied only three series of testing vectors for reason of above limitation. Tables 7.2, 7.3, 7.4 present the fault coverage for each of these series achieved at different starting test vectors. If a fault is not detected by any series of vectors, this fault is obviously treated as not detected. For instance, in the best case, there were 4 such a single faults on „XOR mask" positions 17, 26, 30, 37. It gives 89,7% of fault coverage. Otherwise a fault is detected if it is detected by any of test vector from any of test series.

Testing time of MicroBlaze with the bijective test program is estimated at 8 μs for one test vector. MicroBlaze can be clocked from 100 MHz to 700 MHz in a typical case. Execution of one instruction may require one clock cycle. Only a few instructions require multiple clock cycles. Exhaustive testing accordance to bijective methodology of this processor requires composition of the program with approximately 800 assembler instructions.

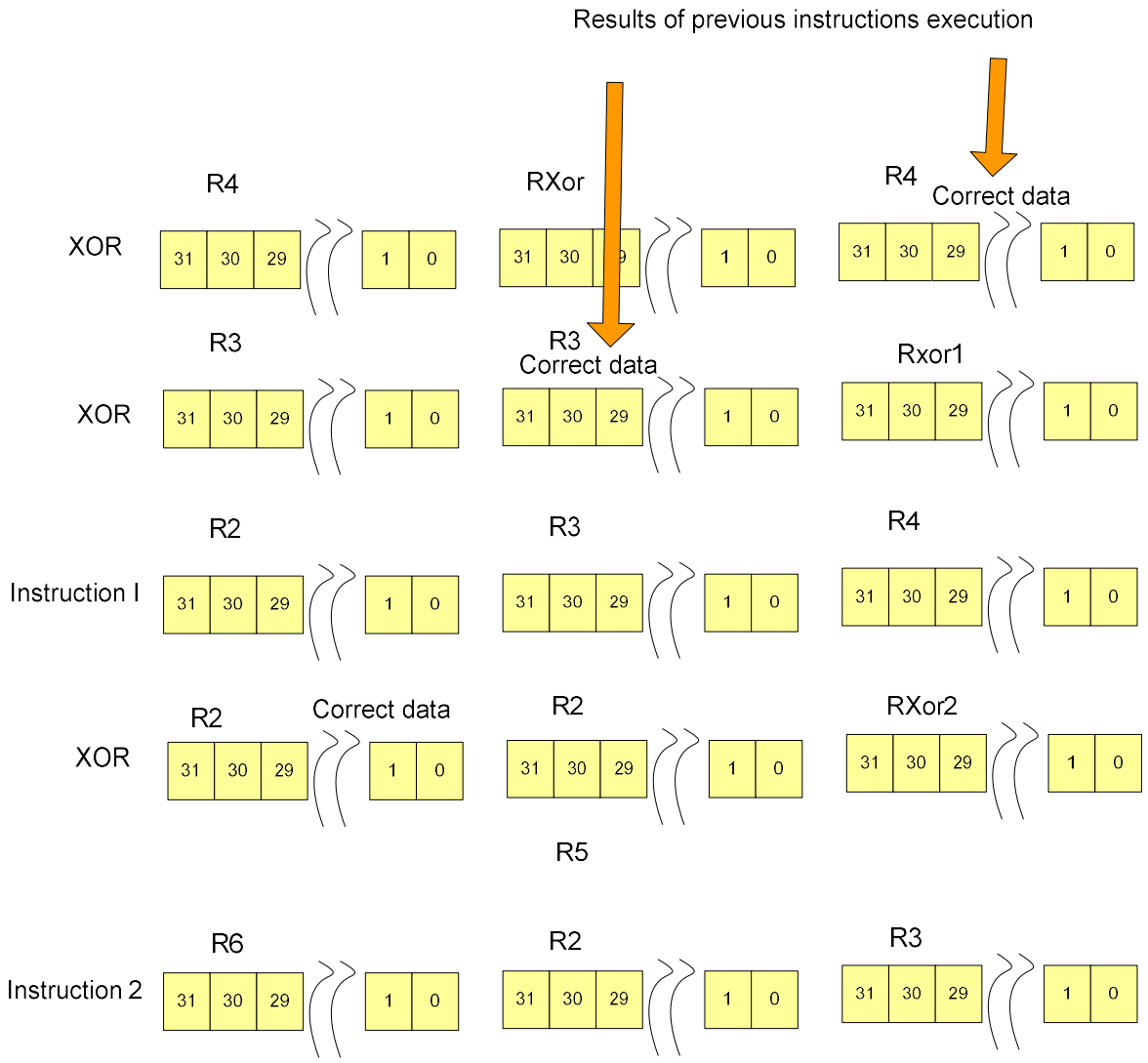


Figure 7.3: The fault injection methodology

Starting vectors for mentioned above testing series are following:

Series I: 55287360 testing vectors, Starting vector: 0x00000000

Series II: 58129920 testing vectors Starting vector:0x000FFFFF

Series III: 70139520 testing vectors Starting vector: 0x7FFFFFFF

Table 7.2: The fault coverage for injected faults in selected bit-locations

Affected registers	Number of injected faults at drown of bits to affect	Fault coverage for Series I of tests	Fault coverage for Series II of tests	Fault coverage for Series III of tests
1 (first)	39	32/39 (82%)	33/39 (84,6%)	28/39 (71,8%)
2 (both)	78	71/78 (91%)	64/78 (82%)	66/78 (84,6%)
1 (send)	39	35/39 (89,7%)	29/39 (74%)	34/39 (87%)

Table 7.3: The results of first random experiment

Affected registers	Number of injected faults at drown of bits to affect	Fault coverage for Series I of tests	Fault coverage for Series II of tests	Fault coverage for Series III of tests
1 (first)	39	21/39 (53,8%)	24/39 (61,5%)	18/39 (46%)
2 (both)	78	52/78 (66,7%)	47/78 (60%)	48/78 (61,5%)
1 (second)	39	25/39 (64%)	17/39 (43,6%)	29/39 (74%)

Table 7.4: The results of second random experiment

Affected registers	Number of injected faults in all possible bit locations	Fault coverage for Series I of tests	Fault coverage for Series II of tests	Fault coverage for Series III of tests
1	195 (5 x 39)	109/195 (55,9%)	122/195 (62,6%)	99/195 (50,7%)
2	390 (10 x 78)	269/390 (69%)	249/390 (64%)	245/390 (62,8%)
1	195 (5 x 39)	141/195 (72,3%)	91/195 (47%)	148/195 (75,9%)

It is obvious, that exhaustive processor testing or evaluation of the test program according to the above methodology is hardly feasible (for 32-bit processor there is a huge number of 2^{32} = 4'294'967'296 test vectors). It should be noted, that these fault injected by „XORing" do not reflect suitable the essence SEU phenomena. All my experiments and discussions about MicroBlaze testing can be treated as merely initial trials. That times, an open question remained: how well the injected faults correspond to the actual SEU -induced faults? In order to resolve this dilemma, I implemented a processor test for the Xilinx PicoBlaze processor core for which the VHDL description of the core is available. Since the PicoBlaze processor core is targeted for FPGA implementation, its VHDL description consists of low-level FPGA functional blocks, LUTs, multiplexers, flip-flops and RAM. These functional blocks are directly mapped to the FPGA resources.

7.4. Summary of the results of researches from bibliography

Table 7.5 presents a summary of the results from bibliography on the subject of microprocessor testing **(Psarakis 2010)**. The researches on the processors whose functionality, construction complexity or performance can be compared with the MicroBlaze were selected. Results of these researches are expressed usually as coverage of injected faults into hardware of given microprocessor / micro-controller in order to compare with achievements of my method. Fault coverage levels can be compared, however, authors usually apply stuck-at fault model, which also does not correspond suitable to SEU phenomena.

Table 7.5: Summary of the results of researches from bibliography in comparison with MicroBlaze

Nr	Author	Tested Processor	Year	Fault coverage [%]
	Wegrzyn	**MicroBlaze**	**2007**	**91 XOR injected faults**
1	Gurumurthy, Vasudevan, and Abraham	OpenRISC 1200	2006	82
2	Psarakis, Gizopolous	OpenRiSC	2010	90
3	Prabhat Mishra	DLX	2004	100, insignificant number of faults 91fun, 177pipelin
4	**Wegrzyn**	**MicroBlaze**	**2007**	**91,1**
5	Bernardi, Sonza	MiniMips 32bit 5pipe	2014	92.67
6	Bernardi, Sonza	OpenRISC	2014	92,7
7	Wen, Wang, and Cheng	OpenRISC1200 (controller and ALU only)	2006	93
8	Gizopoulos et al.	MiniMIPS, OpenRISC	2008	93
9	Chen et al	ARM4 core	2007	94
10	Chen, Dey	The picoJava-II 32bit	2001	95
11	Batcher Papachristou	Different CPUs, DLX (deluxe RISC),	1999	95

Nr	Author	Tested Processor	Year	Fault coverage [%]
12	Chen et al.	Xtensa (ALU only) 32-bit RISC conf./synthes	2003	95
13	Bernardi, Sonza	SPARC Leon I	2004	95 (stack at 0,1)
14	Kranitis et al	Plasma, MIPS R3000	2005	95
15	Psarakis, Gizopolous	MiniMips	2010	95
16	Riefert, Sonza	MIPS-like 5pipelin stg.	2016	95
18	Zhang 2013	OpenRISC	2013	95,5
19	Zhou, Wunderlich	32bit RISC	2006	96
20	Tai-Hua Lu, Chung-Ho Chen	ARMv4	2011	97

7.5. Construction of bijective blocks for MicroBlaze testing

Utilizing gathered knowledge during PicoBlaze testing, it is easy to design bijective program blocks to arithmetic, logic and other instruction testing. Despite the fact that the evaluation of the MicroBlaze test program proved not to be feasible, one can expect similar results as in case of the PicoBlaze constructing its test program accordance to principles developed for PicoBlaze testing. Examples bellow present bijective program blocks to ADDCY, and SUBCY instruction testing (See Listings 7.5.1, 7.5.2, 7.5.3, 7.5.4):

```
16z:    JUMP      comp;     subcy test
ds:     SUBCY     s7, 01;   s7->s7
        JUMP      su;
comp:   COMPARE   s7, 80;   subcy sX, sY test
        JUMP      nc, ds;
su:     SUBCY     s7, 87;
```

Listing 7.5.1: Bijective block to Subtraction with carry testing written in PicoBlaze assembler

```
        Lwi   r8,  r8,  4;    load 4 into r8 register (r8+4)
        Lwi   r9,  r9,  1;    load 1 into r9 register (r9+1)
        Lwi   ra,  ra,  0x70000001;     load 0x70000001 into ra register
        Rsubi r6,  r8,  0;    generation of signed -4 in r6 register
        Mfs   r4,  0;         copy pc in r4
        nop;
16z:    brlid  r4,  4;        pc = pc+4 ; jump to cp compare
 ds:    subc   r7,  r9,  r7
        mfs    r4,  0;        pc in r4
        brlid  r4,  3;        pc = pc+3
cp:     andi   r5,  r7,  0x00000001;
        bne    r5,  s6;
su:     subic  r7,  ra,  r7;
```

Listing 7.5.2: Adequate code written in MicroBlaze assembler

```
        LOAD        sD,  s7;   register change
17z:    JUMP        cp;        addcy test
bcy:    ADDCY       sd,  88;   sD → sD
        JUMP        18z;
cp:     COMPARE     sd,  80;
        JUMP        nc,  bcy;
adcy:   ADDCY       sd,  87;
18z: ----next block
```

Listing 7.5.3: Bijective block to ADD with carry testing written in PicoBlaze assembler

```
        Lwi,   r7,  r7,  5;    load 5 into r7 register
        Rsubi  r6,  r7,  0;    generation of signed -5 in r6 register
        Lwi    rd,  r7,  0;    change of register s7 into D
        Mfs    r4,  0;         pc in r4
        nop;
17z:    brlid  r4,  4;         pc = pc+4, (17z) to cp compare
bcy:    addic  rd,  rd,  0x80000008;   add witth carry immediate value
        mfs    r3,  rmsr;      contains of register msr with carry bit is placed into r3 register
        nop;                   for reason of pipeline stall behavior
        brlid  r3,  4;         jump to pc =pc + 4 to 18z
cp:     andi   r5,  rd,  0x00000001;
        bne    r5,  s6;        branch if r5 !=0 to pc -5 (r6)
        addic  sd,  sd,  0x80000007;      add with carry immediate value
18z:----next block
```

Listing 7.5.4: Adequate code written in MicroBlaze assembler

7.6. Application of LFSR to construction of the MicroBlaze bijective blocks

In case of MicroBlaze SHIFT instructions testing, the same problem; lack of a full cycle appeared, as in case of testing PicoBlaze's SHIFTs. For this reason I have redesigned bijective blocks for MicroBlaze „SHIFTs" testing utilizing the idea of Linear Feedback Shift Register (LFSR), applied before to solve the problem at PicoBlaze „SHIFTs" testing. Architecture and principles of operation of the MicroBlaze „SHIFTs" are similar to PicoBlaze with slight differences as follows. There are nine SHIFT instructions. Six of them have defined number of bits it shifts trough, determined by second input operand register. Remained three SHIFT instructions can shift trough only one bit at one execution. Functionality of few examples of SHIFTs and other instructions utilized in block of program are described as in the User Guide of MicroBlaze. There isn't such instruction as PicoBlaze „TEST". Instead fcomp.eq is applied to detect occurrence of logical „1" on the place determined by the characteristic polynomial of LFSR, which guarantees that a testing vectors generated by the LFSR will form a full cycle. This characteristic polynomial which forms full cycle for 2^{32} of test vectors is given by the formula:

X32 + X22 + X2 + X1 + 1

```
Lwi   r8,   r8,   0x1;  load 0x1 into r8 register (r8+1)
Lwi   r9,   r9,   0x100;
Lwi   rA,   rA,   0x200000;
Lwi   rB,   rB,   0x80000000;
Andi  r1F,  rD,   0x80000000;
Andi  r1E,  rD,   0x200000;
Andi  r1D,  rD,   0x100;
Andi  r1C,  0x1;
fcmp.eq  r7, r1F,  rB; Floating point comparison. If content of r1F and rA equal r7 = 1, otherwise r7 = 0
fcmp.eq  r6, r1E,  rA;
fcmp.eq  r5, r1D,  r9;
fcmp.eq  r4, r1C,  r8;
xor      r6, r6,   r7;
xor      r5, r5,   r6;
xor      r4, r4,   r5;
bsrl1    r4, r4,   0x1F;
SRL      rD, rD;
Or       rD, r3,   r4;
-------Next program block------------
```

Listing 7.6.1: Example of program block LFSR based to Shift Right Logical test

```
Lwi   rB,  rB,  0x5;   load 0x5 into rB register (rB+1)
bsll  rD,  rA,  rB;    Barrel Shift Left Logical. Content of rA is shifted by number specified in rB
Lwi   r8,  r8,  0x1;
Lwi   r9,  r9,  0x100;
Lwi   rA,  rA,  0x200000;
Lwi   rB,  rB,  0x80000000;
fcmp.eq  r7, rD, rB; Floating point comparison. If content of rD and rB equal r7 = 1 otherwise r7 = 0
fcmp.eq  r6, rD, rA;
fcmp.eq  r5, rD, r9;
fcmp.eq  r4, rD, r8;
xor   r6, r6, r7;
xor   r5, r5, r6;
xor   r4, r4, r5;
BSLLI   rD, rD, 0x1; Barrel shift logical left by immediate value 0x1
Or      rD, rD, r4;
--------Next program block--------
```

Listing 7.6.2: Example of program block LFSR based to Barrel Shift Left Logical test

Description of the instructions used in above examples is as follows:

Fcmp.eq Floating-Point Comparison Equal:

Fcmp.eq rD, rA, rB Equal floating-point comparison

Comparison Type		Operand Relationship				
Description	OpSel	(rB) > (rA)	(rB) < (rA)	(rB) = (rA)	isSigNaN(rA) or isSigNaN(rB)	isQuietNaN(rA) or isQuietNaN(rB)
Equal	010	(rD) ← 0	(rD) ← 0	(rD) ← 1	(rD) ← 0 FSR[IO] ← 1 ESR[EC] ← 00110	(rD) ← 0

LWI (Load Word Immediate):

Lwi rD, rA, IMM

Loads a word (32 bits) from the word aligned memory location that results from adding the contents of register rA and the value IMM, sign-extended to 32 bits. The data is placed in register rD.

SRL (Shift Right Logical) :

Srl rD, rA

Shifts logically the contents of register rA, one bit to the right, and places the result in rD. A zero is shifted in the shift chain and placed in the most significant bit of rD. The least significant bit coming out of the shift chain is placed in the Carry flag.

```
rD[0]    ← 0
rD[1:31] ← rA[0:30]
```

Listing 7.6.3: SRL pseudocode

BSLL (Barrel Shift Left Logical):

Bsll rD, rA, rB

Shifts the contents of register rA by the amount specified in register rB and puts the result in register rD. The mnemonic bsll sets the S bit (Side bit). If the S bit is set, the barrel shift is done to the left. The mnemonics bsrl and bsra clear the S bit and the shift is done to the right. The mnemonic bsra will set the T bit (Type bit). If the T bit is set, the barrel shift performed is Arithmetical. The mnemonics bsrl and bsll clear the T bit and the shift performed is Logical.

```
if S = 1 then
  (rD) ← (rA) << (rB)[27:31]
else
  if T = 1 then
    if ((rB)[27:31]) ≠ 0 then
      (rD)[0:(rB)[27:31]-1] ← (rA)[0]
      (rD)[(rB)[27:31]:31] ← (rA) >> (rB)[27:31]
    else
      (rD) ← (rA)
  else
    (rD) ← (rA) >> (rB)[27:31]
```

Listing 7.6.4: BSLL pseudocode

BSLLI (Barrel Shift Left Logical Immediate):

Bslli rD, rA, IMM

The first three instructions shift the contents of register rA by the amount specified by IMM and put the result in register rD. The mnemonic bsll sets the S bit (Side bit). If the S bit is set, the barrel shift is done to the left. The mnemonics bsrl and bsra clear the S bit and the shift is done to the right.

The mnemonics bsrl and bsll clear the T bit and the shift performed is Logical.

```
if S = 1 then
  (rD) ← (rA) << IMM
else
  if T = 1 then
    if IMM ≠ 0 then
      (rD)[0:IMM-1] ← (rA)[0]
      (rD)[IMM:31] ← (rA) >> IMM
    else
      (rD) ← (rA)
  else
    (rD) ← (rA) >> IMM
```

Listing 7.6.5: BSLLI Pseudocode

In conclusion, the proposed by me fault injection method by „XORing" do not reflect suitable the essence of SEU phenomena. Other authors usually apply stuck-at fault model too, which also is completely inadequate to SEU phenomena. I was performing such experiments rather in order to estimate their time and to consider experimental set of equipment. There exist possibility of evaluation MicroBlaze test program using SecretBlaze processor core, because SecretBlaze functionality is almost the same as MicroBlaze. But it would take much more time as in case of PicoBlaze. Thus, all my experiments and discussions about MicroBlaze test program evaluation can be treated as merely initial trials.

Basic principles of my solution can be utilized for testing other microprocessors. It is one of the main benefits of my method. There is no reason to expect significantly different test results for other processor cores dedicated for FPGA. When any processor is implemented in the same FPGA family, its structural level VHDL description looks very similar for any model of such processor, independently from its high level architecture. Every processor core will be decomposed to the same basic low level primitives, and in every case logical functions implemented by the same LUTs will looks similar and will meet the same logical redundancies described in Table 9.2.

8. Evaluation of the test program

8.1. Evaluation scheme

When selecting the processor core for the case study, an important question is how the developed solution actually will be evaluated? In order to determine the fault coverage of SEU-induced faults some means of fault injection must be provided. As an alternative to the statistics-based radiation tests **(Lesea 2005)** I am looking for a simulation-based solution. Simulation-based fault injection is made difficult by the lack of commercial tools that would allow the user to alter the FPGA configuration once it is translated from the HDL source into the target FPGA platform. However, one possible solution is to insert faults prior to the test program's execution. This approach is general (i.e. it can be applied for different processor cores), but it also has some limitations and requires rather a lot of manual interference. Alternatively, the HDL description can be used to model the system and simulate its performance as well as the faults within the system.

The same system can be described in HDL in a number of different ways. Three extreme cases are: 1) high-level algorithmic descriptions (irrespective of the target system's structure), 2) RTL (register transfer logic) a level description in which functionality is described at the level of operations among the actual system components (i.e. registers), 3) structural VHDL – where gate level description of the module is given. A high-level description of a core does not provide enough details for realistic HDL modeling of the SEU-induced faults. On the other hand, soft-processors like the Xilinx PicoBlaze are developed for FPGA implementation and their structural description is given.

The basic HDL entities in a description of the Xilinx PicoBlaze processor core are RAMs, LUTs, multiplexers and flip-flops. SEU-induced faults can alter the contents of RAMs, LUTs or flip-flops or they can modify the connections between these functional blocks. RAM and flip-flop content changes are of a transient nature and can be modified (i.e. restored to a fault-free value) during normal system operation. These faults may only be detected with an online functional test, specific to the target

application and hence are not the subject of this investigation. My goal is to detect permanent faults in the configuration of the processor core.

According to **(Wegrzyn 2014 A)** PicoBlaze HDL descriptions reflect the FPGA structure in order to efficiently use the FPGA resources. This allows precise modeling of the faults and their automated fault injection. For each simulated fault, an appropriate HDL file is generated. All the fault injection campaign and analyze of their effects is perform by a Perl script in an automatic way. The faults in an HDL description of the processor are simulated by modifying the individual functional blocks. For each functional block a HDL models represent behavior of SEU. The HDL model should actually reflect the change of configuration as a consequence of the SEU effect. All the details regarding functionality modification of individual blocks by SEUs are described in further chapters.

I modeled the faults as described in **(Dutton 2009)** and injected them into the VHDL description of the processor core. The following types of LUT faults were modeled:

- faults in the configuration memory holding the values of the LUT,
- faults affecting the input signals (i.e. address decode of the LUT),
- faults affecting the LUT outputs.

8.2. Environment composition for experiments

The developed experiments are targeted at testing fault susceptibility of application programs running on a micro-processor implemented within FPGA. The general idea is to use appropriate microprocessor simulator which accepts its specification in HDL language, correlates it with the targeted FPGA, performs simulations of executing provided programs (in assembler) and allows analyzing the behavior of the tested application (e.g. program results) in this environment. These assumptions are performed by two simulators: Cadence NC VHDL and Mentor Graphics ModelSim.

8.3. Proposed technique for dedicated FPGAs fault injection

The generation of the fault descriptions was implemented as a Perl script **(Wegrzyn 2014 A), (Wegrzyn 2009)**. All the instances of LUTs contained in functional blocks of the processor are described in the VHDL code. For each LUT instance its initialization parameter is investigated and the list of the initialization parameters describing all the SEU-induced faults as well as all the stuck-at faults at the LUT inputs and outputs are generated. For some LUT instances it is possible that a single bit change of a LUT content manifests itself as a stuck-at fault. In such a case a duplicated stuck-at fault description is excluded. In a similar way the stuck-at faults at the LUT inputs as well as the stuck-at faults at the LUT output can also be modelled by modifying the contents of the LUT configuration. In such a case a duplicated stuck-at fault description is omitted.

An example **(Wegrzyn 2014 A)** of a modelled fault is shown in Figure 8.1. The VHDL description of a LUT implementing a circuit generating an internal processor signal move-group is shown in Figure 8.1a. and the corresponding truth table in Figure 8.1c. The LUT input signals I0-I3 relate to the specified bits (in brackets) of the instruction code (in the instruction register). The implemented logic function is defined by the initialization parameter (INIT) assumed as X"7400", i.e. hexadecimal code related to a vector comprising bits of concatenated columns O(I3 = 1) and O(I3 = 0). The most significant bits of O(I3 = 1) and O(I3 = 0) bytes relate to the last row of the table. The SEU-induced fault of a LUT typically manifests itself as a change of one bit of the LUT, thus modifying the Boolean function it implements. For further explanation, to a LUT inputs (denoted from I0 to I3) may be applied: an bit of the instruction op-code, value of any bit from register, a bit of immediate data value, any intermediate signal of processor, constant value, etc.

```
move_group_LUT: LUT4
generic map (INIT => X"7400")
 port map( I0 => instruction(14),
           I1 => instruction(15),
           I2 => instruction(16),
           I3 => instruction(17),
            O => move_group );
```

(a)

```
move_group_LUT: LUT4
generic map (INIT => X"7480")
 port map( I0 => instruction(14),
           I1 => instruction(15),
           I2 => instruction(16),
           I3 => instruction(17),
            O => move_group );
```

(b)

Fault-Free LUT						LUT with changes				
I3	I2	I1	I0	O		I3	I2	I1	I0	O
0	0	0	0	0		0	0	0	0	0
0	0	0	1	0		0	0	0	1	0
0	0	1	0	0		0	0	1	0	0
0	0	1	1	0		0	0	1	1	0
0	1	0	0	0		0	1	0	0	0
0	1	0	1	0		0	1	0	1	0
0	1	1	0	0		0	1	1	0	0
0	**1**	**1**	**1**	**0**	→	**0**	**1**	**1**	**1**	**1**
1	0	0	0	0		1	0	0	0	0
1	0	0	1	0		1	0	0	1	0
1	0	1	0	1		1	0	1	0	1
1	0	1	1	0		1	0	1	1	0
1	1	0	0	1		1	1	0	0	1
1	1	0	1	1		1	1	0	1	1
1	1	1	0	1		1	1	1	0	1
1	1	1	1	0		1	1	1	1	0

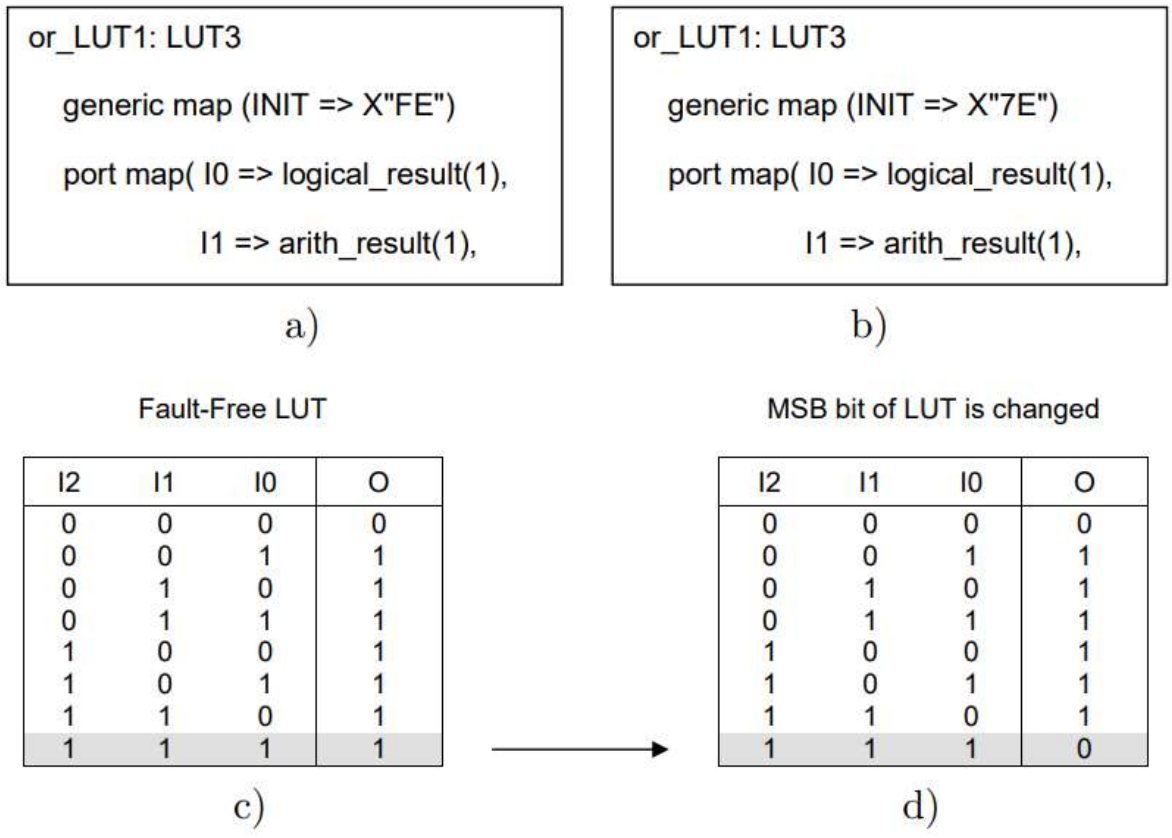
c)

Figure 8.1: Fault effect related to the change of one bit (X"7400") → X"7480") in LUT4
a)VHDL description of fault-free four-input circuit. b) VHDL description
of four-input circuit with a fault. c) truth tables for fault free and faulty LUT

Let us assume that the 8th bit of the LUT column O(I3 = 0) has been changed (truth table in Figure 8.1c with marked false value as bold underlined 1). This fault can be modelled in VHDL description changing the initialization parameter (INIT) from X"7400" to X"7480". Similarly, I can model stuck-at faults on inputs or outputs. For example a stuck-at-1 fault at input I3 in the considered LUT is modelled by INIT = X"7474", i.e. column O(I3 = 0) assumes the value of column O(I3 = 1). Stuck-at-1 fault at output O is modelled by INIT = X"FFFF" (all LUT memory entries equal to 1). The considered LUT

(LUT4 in VHDL description) in Figure 8.1a. relates to the decoder for the control of the program counter and CALL/RETURN stack. Having analysed the effect of the simulated fault (INIT = X"7480") I have found that it resulted in erroneous decoding of SUB or SUBCY instructions as RETURN or JUMP with unknown address locations, so the program did not terminate correctly.

Other example of a modeled fault is shown in Figure 8.2. The HDL description of a LUT implementing a three-input OR gate is shown in Figure 8.2a. and the corresponding truth table, in Figure 8.2c. The SEU-induced fault of a LUT typically manifests itself as a change of one bit of the LUT, thus modifying the Boolean function it implements. Let us assume that the most significant bit of the LUT has been changed, as shown in Figure 8.2d. The fault can be modeled by changing the initialization parameter (INIT), as shown in Figure 8.2b.



I2	I1	I0	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

I2	I1	I0	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

(a) HDL description of fault-free three-input OR gate, (b) most significant bit of the LUT is changed (X"FE" → X"7E")

Figure 8.2: Fault effect related to the change of one bit (X"FE") → X"7E") in LUT3

In a similar way the stuck-at faults at the LUT inputs as well as the stuck-at faults at the LUT output can also be modeled by modifying the contents of the LUT configuration. An example of a stuck-at-0 fault of input I2 is depicted in Figure 8.3. The contents of the LUT in Figure 8.3a. are changed by initializing the parameter (INIT), as shown in Figure 8.3b. The truth tables corresponding to the fault-free LUT and the stuck-at-0 fault of input I2 are shown in Figure 8.3c. and d.

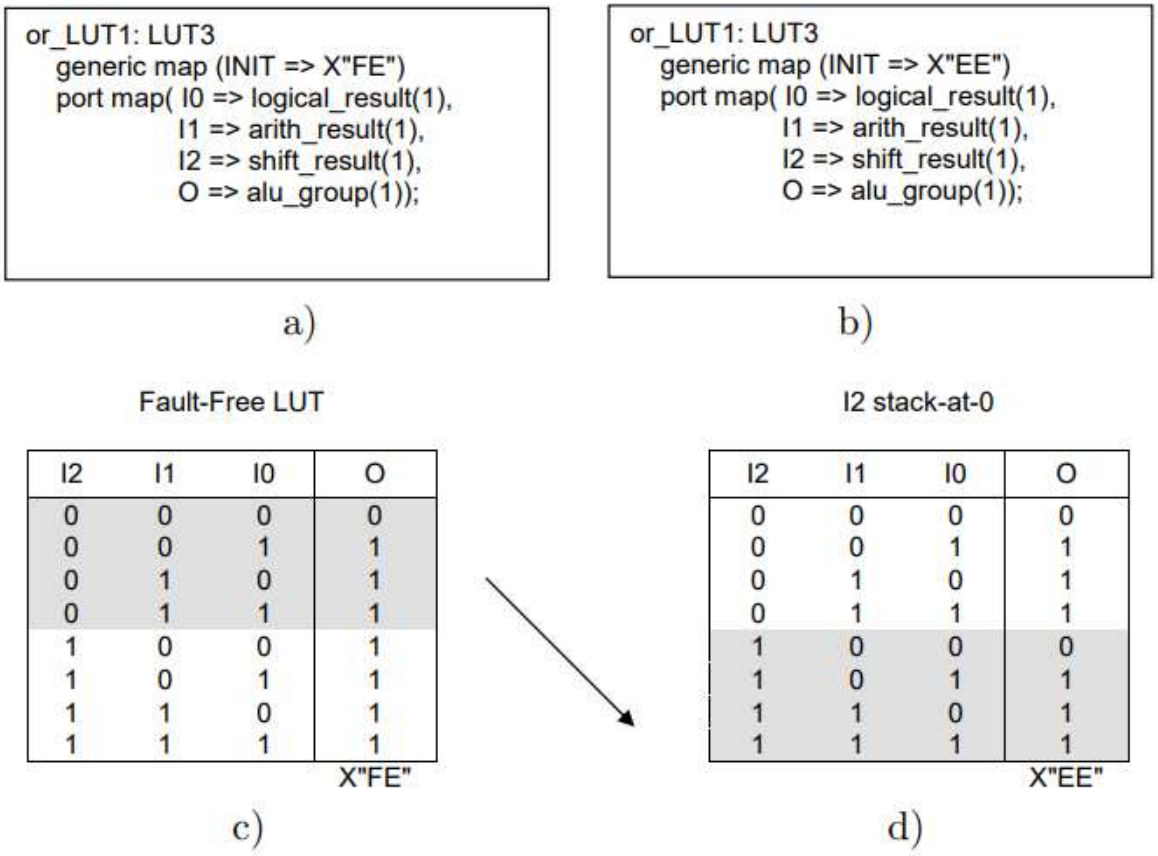


Figure 8.3: Fault effect related to the change of one bit (X"FE") → X"EE") in LUT3
a) HDL description of fault-free three-input OR gate, (b) most significant bit of the LUT ischanged (X"FE" → X"EE"). c) truth table for fault free d) truth table for faulty LUT

8.4. Fault injection implementation

At the time when I started my experiments, no tool for fault injection, integrated with ISE or ModelSim existed. Hence, I took up the challenge of development a novel environment for injecting faults on the basis of CADENCE software tools. The fault injection was implemented in two steps:

- a description of the faults,
- an HDL simulation of the system with generated faults.

All fault descriptions had been placed in a file of faults, and then read by the Perl script. The set of faults is developed in this way, that content of LUT is altered only on one bit or on many bits when stuck at fault is injected. This lead to slight modification of a logical function realized by LUT. Such faults are more difficult to test. During fault simulation the generated „faulty" initialization parameters were applied one by one to the HDL description of the Xilinx PicoBlaze processor core **(Wegrzyn 2009)**, **(Wegrzyn 2014 A)**. A modified HDL description was used, running the test sequence with different input vectors and the results were recorded for a later offline evaluation. A Cadence NC VHDL simulator running on a Sun Fire V240 server and then i7 core Intel was used for the HDL. An example of Cadence tool window is presented in Figure 8.4.

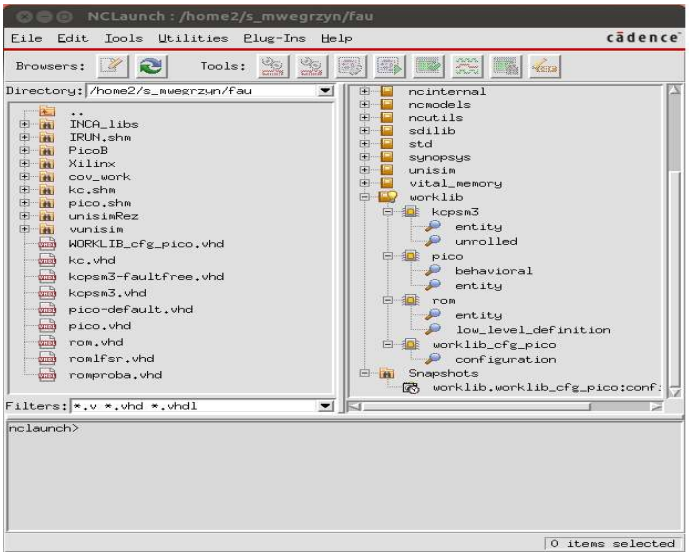


Figure 8.4: NC Launch Cadence tool

Taking into account a large number of considered faults I used special script which automates the processes of loading new configuration, running the application and storing results. Having injected the specified number of fault, I compared the results with a benchmark patterns using an additional script which qualified fault effects and generated summarized statistics.

I needed to modify the original VHDL description to enable the Perl script fault injection. The PicoBlaze hardware is generated using VHDL loops „for" which create more instances of the same sub-circuits (most loops replicate 8 times bit slices of some logical blocks). To make accessible all these instances to the fault injector, I have „unrolled" all „for" loops (explicit code blocks embedded in VHDL description). The unrolled VHDL description results in about 3000 lines of code. In this way the Perl script has direct access to every line of the hardware description. The fault injection is implemented by reading line by line of this unrolled VHDL description by the Perl script. The script was looking for proper strings (description of INIT parameters) in the code and then changes values of these parameters. After every change was completed, simulation is started by the same script. Figure 8.5 presents execution of the Perl script to fault injection in a linux terminal.

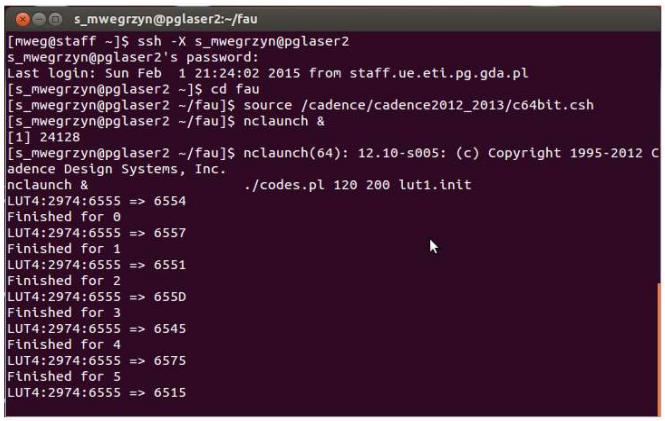


Figure 8.5: Execution of the Perl script in a linux terminal

8.4.1. Description of the PicoBlaze structural VHDL and scripts programs

It is also possible to trace effects of individual fault injection even on the signal levels inside selected internal logic circuits, e.g. an output of some flip-flop. For the PicoBlaze processor instance I have identified 1804 single bit faults related to used LUTs. The Xilinx PicoBlaze processor is a small 8-bit microprocessor, it has 1K of program instructions, sixteen 8-bit registers, 256 input and 256 output ports, a 64-byte internal scratchpad RAM and a 31-location stack. The original VHDL description of the processor core consists of about 1500 lines of code.

In the VHDL description I can distinguish 14 modules, I have placed functions of these modules in Table 8.1, related VHDL description lines (beyond these lines there are some initialization, comments and control lines) and the number of functional FPGA elements used in these modules (CLB – logical blocks, LUT– configuration tables, FF – flip-flops, MUX – multiplexers and XOR circuits). The presented parameters give some view on the PicoBlaze microprocessor complexity. Detect-ability of faults inside of particular blocks for the bijective program with LFSR solution is presented in Table 8.2. Result of merely 84 undetected faults has been achieved in the case when I had neglected all the faults related to I/O ports. So only 1603 among injected faults were taken into consideration.

Table 8.1: Detect-ability of faults inside of particular blocks of PicoBlaze. Bijective merely program

PicoBlaze block name	# lines	# CLB	# LUTs	# Injected faults	# Detected faults	# Undetected faults
Basic control	305 - 326	1	1	2	2	0
Interrupt logic	343- 397	2	3	47	4	43
Dec. control PC CALL/RETURN stack	419 - 468	3	6	104	67	37
ZERO/CARRY flags	484 - 630	6	11	179	140	39
Program Counter	698 - 948	10	20	320	243	77
Register bank and second operand select.	1208 - 1467	5	10	157	155	2
Memory storing function	1493 – 1508	5	2	35	30	5
Logical op.	1718 - 1861	5	9	233	225	8
Shift/ Rotate op.	1888 - 2079	6	11	176	148	28
Arithmetical op.	2017 - 2396	6	11	156	150	6
ALU MUX	2418 - 2641	9	17	217	133	84 in/out
R/W strobes	2674 - 2691	2	3	52	9	43
CALL/RETURN stack control	2964 - 3075	6	5	126	66	60
SUM		73	115	1804	1372	432

Table 8.2: Detect-ability of faults inside of particular blocks of PicoBlaze.

Bijective program + LFSR solution

PicoBlaze Block name	**# lines**	**# CLB**	**# LUTs**	**# Injected faults**	**# Detected faults**	**# Undetected faults**
Basic control	305 - 326	1	1	2	2	0
Interrupt logic	343 - 392	2	3	47	9	38
Dec.control PC CALL/RETURN stack	414 - 463	3	6	104	67	37
ZERO/CARRY flags	479 - 625	6	11	179	147	32
PC	693 - 944	10	20	320	259/309	61(for LFSR) or 11(PC extended)
Register bank/ second and operand selection	1208 - 1467	5	10	157	157	0
Memory storing function	1493 - 1508	5	2	35	35	0
Logical op.	1713 - 1856	5	9	233	232	1
Shif /Rotate op.	1888 - 2079	6	11	176	162	14
Arithmetical op.	2107 - 2396	6	11	156	152	4
ALU MUX	2418 - 2641	9	17	217	133	84 (in/out)
R/W strobes	2674 - 2691	2	3	52	9	43
CALL/RETURN stack control	2964 - 3075	6	5	126	66/104	60 (for LFSR) or 22 (stack extended)
SUM		73	115	1804	1427/ 1518	286/84

8.4.2. Description of auxiliary scripts

After simulation on CADENCE NC VHDL tool I have obtained, the output file of 1804 x 256 results for every 256 input test vectors for every consecutive of 1804 injected faults. The file looks as in example in Figure below. It can be noticed, that columns of order and results numbers are separated by the line of description of every injected fault. At beginning the name of a LUT Xilinx primitive is placed, then a line number where a fault is injected in the PicoBlaze VHDL description, and finally an example EAAA => EEAA gives us information, that INIT parameter, which determines logical function of LUT4 Look-up table is changed on one bit position A = 1010 into E = 1110.

```
LUT4:382:EAAA => EEAA
00   05
01   06
02   07
03   08
……
……
FF   04
LUT4:382:EAAA => E2AA
00   05
01   06
02   07
……
……
FE   03
FF   04
```

The reference file of 256 results for 256 input test vectors has been generated by execution the test program on the correctly operating PicoBlaze. The test program is written in PicoBlaze assembler, and then placed together with starting input vector in a ROM memory dedicated for the PicoBlaze processor. Next input vectors are generated in a loop and the test program is executed 256 times, every time with a different test vector. The program is executed on ModelSim simulation tool. Such

the VHDL description in dedicated to Xilinx FPGAs. The text file of output results is generated by proper functions of VHDL.

A need to check the bijective property of these results appears, and next to calculate detect-ability of test program. For this purpose several script-programs were developed. The first of them checks exclusively if any of results is not repeated in the result vector column. It is lead on fault-free output result file obtained from simulations of test program with the ModelSim simulator. This is the file of exclusively 256 reference results. Figure 8.6 presents the interface of the ModelSim.

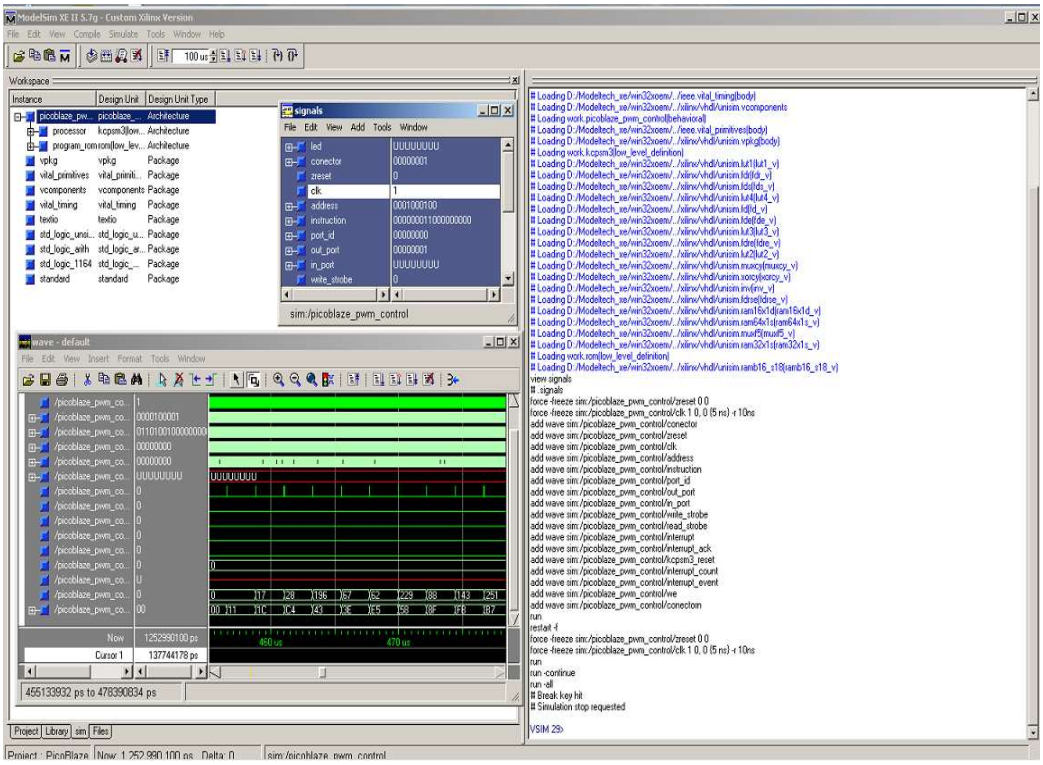


Figure 8.6: The ModelSim interface

The second script program is written for calculation of the result fault coverage of the test program. It works in the following way; compare 1804 times the reference fault-free result file with the output file of 1804 x 256 results (lines as in example above) obtained from the CADENCE simulator. If there is one or more differences on any resultant number, a fault is detected and a fault counter is increased by 1. The script cuts the headers lines of fault description among every next comparison as in example (LUT4 : 382 : EAAA => EEAA) -line of the fault description.

Other script programs have been written in order to investigate different optimization strategies (see Section 6). This program is composed of three blocks. The first one reads a reference file and the file after VHDL simulation on CADENCE with injected consecutively faults, and fills a two dimensional array of detected and undetected faults and test vectors which detected a specified fault (vtab[f][i], where f is a fault number from the range from 0 to 1803 and „i" is a number of test vector from range 0 to 255). A sum of vectors, which detected every fault is calculated in a next loop for every individual fault. Finally faults are sorted and printed to resultant file with the lowest-order faults first.

The program prints out on file headers describing these the hardest to detect faults according to the following example:

„Nr of fault" f. description: LUT4:382:EAAA => EEAA, „ vector which detected the fault"
„Nr of fault" f. description: LUT4:382:EAAA => FAAA, „ vector which detected the fault"
…
„Nr of fault" f. description: LUT4:382:EAAA => CAAA, „ vector which detected the fault"

As a conclusion to the chapter, I would like to emphasize that one of the most important novelty introduced hereby is a different model of injected faults. This functional model of such faults differs considerably from the conventional stuck-at fault model due to the fact that SEU-induced faults affect FPGA configuration SRAM especially logic implemented by the look-up tables (LUT) in this manner that the logic function is arbitrarily changed. No one before, had introduced such a fault model based on principles implemented by me. I have reflected SEUs induced faults in LUTs accordance to nature of these physical phenomena in semiconductors and range of their appearance as described in bibliography of subject (**Gaspard 2017**), **(Rebaudengo 2002 A), (Rebaudengo 2002 B)**. Benefit of my novelty is double, because in this way it is possible to model natural SEU faults in LUTs, which lead to different implementations of logical functions as these intended. Second benefit is, that these faults can be interpreted in particular cases as a stuck-at „0" or stuck-at „1" faults at inputs or output of LUTs, so the FPGA routing resources are also tested. Stuck-at fault interpretation is needed here also for purposes of FC comparison with other solution, where authors use merely the stuck-at fault model.

The proposed model is very closed to real model, thus when combined with optimization heuristics, it significantly reduces the number of test vectors required to achieve FCmax. Other resources as i.e. switches, programmable interconnections are not available from Xilinx programming tools level or even access is protected against people outside the corporation.

An evaluation case study of a functional test for the PicoBlaze processor core was performed on a generalised fault model, including both stuck-at faults and functional faults in LUTs, which are more difficult to detect. The achieved fault coverage confirms the efficiency of the proposed bijective approach. Despite the fact that faults induced by SEU in LUTs are harder to detect, I have obtained results comparable to other authors of publications, which utilized only stuck-at fault model.

9. Problem of faults masking

9.1 Analysis of detected faults

After experiments of the PicoBlaze test programs evaluation have been finished, it is still visible in Tables 5.4.1, 6.3.13, 8.1, 8.2 that achieved FC is less than 100%. Hence, a question appears what the reason is? The answer can be obtained on the basis on detailed analysis of interaction between test program written in assembler together with suitable data and PicoBlaze hardware. Structural VHDL description and effects of injected faults are described in chapter 8. In this chapter my investigations focus on demonstration how it happens, that a fault can be detected or not. Particularly which assembler instructions with deterministic data must be executed to detect given fault or group of faults.

LOGIC INSTRUCTIONS

The logic group of instructions provide bit-wise logic operations on two operands. These operations can be selected by a multiplexer. An important observation is, that value “6E8A” of INIT parameter represents four input function which fits for a Look-up table, and digits 6, E, 8, A determine adequately XOR, OR, (AND, TEST) and LOAD operations. Accurate bit representation is shown in Table D-1 „PicoBlaze Instruction Codes” (see attachment PicoBlaze User Guide on CD). Architecture of PicoBlaze logical instructions is organized as presented in Figure 9.1 bellow:

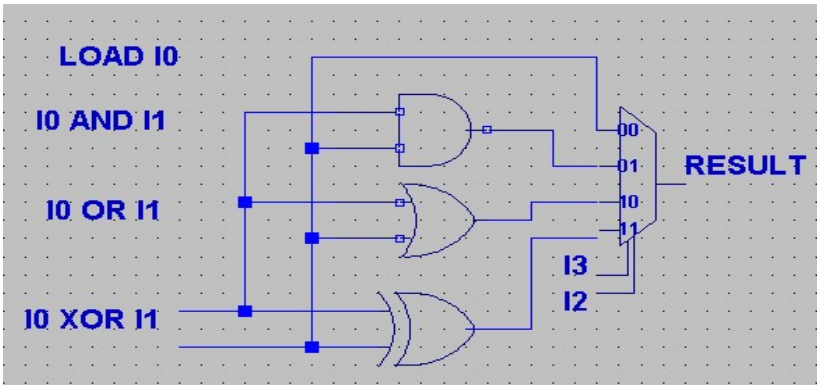


Figure 9.1: Architecture of PicoBlaze logical instructions

Correct logical function implemented by LUT, related to value of INIT = 6E8A is as present:
O = ((I0 I1 (!I3)) + (!I0 I1 I3) + (I0 (!I1) I3) + (I0 (!I2))), where I0…I3 are LUT's inputs.

First example shows what happens after modification of the INIT parameter in an instance LUT4 : 6E8A => 6E82. It is visible, that operation of LOAD instruction was changed to a different unintentional logical function on 5th bit :

F(LUT4) = ((I0 I1 I2 (!I3)) + (I1 (!I2) I3) + (I0 (!I1) I3) + ((!I0) I1 I3) + (I0 (!I1) (!I2))).

This fault results in additional logical sum component and additional variables I2, (!I1) which constitute sum components (marked in red). Original VHDL description of this sub-block is presented in Figure 9.2a. Its description after change of the ,,INIT" parameter is shown in Figure 9.2b.
Additionally, based on information about which signals are assigned to LUT inputs (see PicoBlaze vhdl on CD), I have investigated, that in order to detect this fault it is necessarily to put into second operand number 0x20 (5th bit = 1). In this case when we have ,,1" in first operand register on 5th bit and when we load ,,1" from second operand we will obtain unexpected result ,,0" of LOAD operation on this bit. (LOAD operation has I3 = 0, I2 = 0, see ,,PicoBlaze Instruction Codes"). It is caused by additional I2 ingredient in the first sum. Similarly in case of additional variable (!I1) of logic sum; when we execute LOAD sX, 0x20 with register sX = 0x20, we will obtain unexpected result 0 on 5th bit. In this way this error is detected. Additional sum component (I1 (!I2) I3) forms redundant one argument ,,OR" function, and it has no effect.

```
logical_LUT5: LUT4
   generic map (INIT => X"6E8A")
  port map(  I0 => second_operand(5),
     I1 => sx(5),
     I2 => instruction(13),
     I3 => instruction(14),
     O => logical_value(5));
```

(a)

```
logical_LUT5: LUT4
   generic map (INIT => X"6E82")
  port map(  I0 => second_operand(5),
     I1 => sx(5),
     I2 => instruction(13),
     I3 => instruction(14),
     O => logical_value(5));
```

(b)

Figure 9.2: Examples of correct LUT output function and modified one

The next example describes an error in LUT related to AND instruction. When the INIT parameter is changed into „6ECA", the LUT function takes form:

O = ((I1 I2 (!I3)) + ((!I0) I1 I3) + (I0 (!I1) I3) + (I0 (!I2)))

One sum component is modified and constitutes faulty operating „AND" function. In order to detect this fault it is enough to execute AND sX, 0 instruction, with value of register sX = 0x20. The faulty result will 0x20, instead 0 (incorrect 1 on 5th bit).

Other examples, where INIT parameter is modified into „EE8A" and the LUT function is changed into: F(LUT4) = ((I0 I1) + (I1 I3) + (I0 I3) + (I0 (!I2)) is placed in section of LUT generating „XOR" operations. Such function can easily generates wrong results independently from executed assembler instruction. To detect this fault enough is to execute i.e XOR sX, 0x20, where sX = 0x20. Faulty result will 0x20 instead 0.

The last fault affects „OR" instruction section. The LUT function after modification INIT parameter into „668A'' is as present:

O = ((I0 I1 (!I3)) + (I0 (!I1) I3) + (!I0 I1 I3) + (I0 (!I2) (!I3)))

Will generate wrong result „0" in case when we execute e.g. OR sX, 0x20, where sX = 0x20.

Due to the lack of coverage of all situations, when „OR" function takes logical „1".

There are undetectable faults due to some kind of logical and hardware redundancies as described in chapter 9.2.

9.2 Analysis of masked faults

After test program execution, there remained 286 undetected faults of all simulated 1804 SEU or stuck-at faults. In this chapter the problem of faults masking is considered for every microprocessor block. Effects of fault injection are presented and classified into five categories. The HDL description of the PicoBlaze is divided into such blocks as:

1. Fundamental Control unit which defines T-state and internal reset. (Lines of VHDL description 304-327) Faults 0 up to 1,
2. Interrupt logic (Lines 340 to 409) Faults 2 up to 48,
3. Decoder for the control of the program counter and CALL/RETURN stack (Lines 417 to 472) Faults 49 up to153,
4. The ZERO and CARRY Flags (Lines 479 to 670) Faults 154 up to 332,
5. The Program Counter (Lines 681 to 1196) Faults 333 up to 652,
6. Register Bank and second operand selection (Lines 1211 to 1476) Faults 653 up to 809,
7. Store Memory (Lines 1489 to 1512) Faults 810 up to 845,
8. Logical operations (Lines 1704 to 1861, AND, OR, XOR, LOAD, TEST, Includes pipeline stage to form ALU multiplexer including decode) Faults 846 up to 1079,
9. Shift and Rotate operations (Lines 1873 to 2085, Includes pipeline stage used to form ALU multiplexer including decode) Faults 1080 up to 1255,
10. Arithmetic operations (Lines 2098 to 2400, ADD, ADDCY, SUB, SUBCY, COMPARE. Includes pipeline stage used to form ALU multiplexer including decode) Faults 1256 up to 1393, Faults 1394 up to 1411,
11. ALU multiplexer (Lines 2409 to 2646) Faults 1412 up to 1628,
12. Read and Write Strobes (Lines 2656 to 2696) Faults 1629 up to 1680,
13. CALL/RETURN stack (Lines 2709 to 3075) Faults 1681 up to 1804.

Statistics of undetected faults in above defined functional blocks is presented in Table 9.1.

Table 9.1: Statistics of undetected faults in individual blocks of PicoBlaze

Number of processor block	**Number of injected faults**	**Number of undetected faults**	**Undetected faults [%]**
1. Fundamental Control Unit	2	0	0
2. Interrupt logic	47	38	80,86
3. Decoder for control PC, CALL/RETURN stack	104	37	35,6
4. ZERO, CARRY Flags	179	32	17,9
5. Program Counter	320	61 (for LFSR) or 11 (PC extended)	19,1/3,4
6. Register Bank and second operand selection	157	0	0
7. Store Memory	35	0	0
8. Logical operations	233	1	0
9. Shift and Rotate operations	176	14	8
10. Arithmetic operations	156	4	2,6
11. ALU multiplexer	217	84 (inputs)	38,7
12. Read and Write Strobes	52	43 (in/out)	82,7
13. CALL/RETURN stack	123	60 (for LFSR) or 22 (stack extended)	48,8 /17,9
SUM	1804	286 (with in/out) or 84 without (in/out)	15,8 / 5,24

These experiments targeted to evaluate Fault Coverage (FC) separately for program counter (PC), call return stuck or data flow through bijective blocks. In the case of the PC all the program was duplicated three times. This results in about 1000 instructions in total, and total FC increased by 11%, but of course three times longer simulation time. In case of shifts it was easier to work upon improvements with shorter program composed merely with : „LFSR shifts" bijective blocks or „LSFR shifts" and possibly bijective blocks to arithmetic and logic instruction testing. In such a case remains 61 % undetected faults in the PC block.

As mentioned in Section 5.4.2, a testing engineer encounters problem related to masking of faults **(Bolchini 2007)**. These problems have their source among others in:

- overlapping of flags generated by different instructions, operation of different instructions on the same registers and data. In this way flow of information can be disturbed.

- nature of SHIFT instruction operating, (non-exhaustive set of numbers can be generated using exclusively a shift instructions without any special solutions as LFSR.)
- masking the flow of information related to processor hardware construction as pipeline delays, hardware redundancies, simplifying the construction of individual sub-blocks of processor, etc.

Since the Xilinx PicoBlaze processor core is designed for FPGA implementation and its HDL description consists of low-level FPGA functional blocks that are directly mapped to the FPGA resources. Carrying out detailed analytical proves of detectability is very time consuming, but concurrently very interesting. That's why I carried out analysis of certain quantity of faults which are detected by testing programs. These researches revealed several reasons why certain number of faults still remains undetected. These are both other than intended logical functions, some sorts of logical and hardware redundancies **(Renovell 2000 B)**, **(Renovell 2001)**, **(Renovell 2000 C)** and vice versa reductions, described in details bellow. In order to approximate the importance of such undetected errors for the development of the test method, a statistics have been created. After fault is injected, logical function implemented by a LUT is modified in few ways as it is further described. Additional component (redundant) of logic sum is often created, its logical value is every time equal „0" or this additional component is unable to disturb correct execution of instructions. Additional variables of logic product constituting the logic sum are created, some components of logic sum are reduced, some variables of logic product constituting the logic sum are reduced too, and eventually one or more variables constituting components of logic sum are changed. Often such a problem appears, that combined logical value of several LUT functions remains the same after change of a single LUT INIT parameter, despite the fact that local equation (logic function) is different. Such a problem is described upon example of LOAD, and select parity function at TEST instruction. Other analyzed problems are such that, created additional component of logic sums or additional variables of logic product take every time value „0" for reasons of other variables inside a block as in an example of arithmetical operations described below or by themselves. Sometimes additional components or function variables are unable to disturb correct instruction execution. Vice versa reduced components and variables in logic sums are sometimes unable to disturb correct program execution. As mentioned above, this case is described based on example of LOAD instruction, where propagation of set flag by this instruction as result of fault is impossible for reason of construction of other hardware.

Additional component of logic sum or additional variables of logic product constituting the logic sum, required usage of specified instruction with determined arguments in order to detect them. This instruction should be used extra outside the main algorithm. This depends on specific situation. Additional components can be logically redundant or not.

The number of undetected faults of all the categories are collated in Table 9.2, respectively for every functional block of the microprocessor. A few most interesting cases of undetected faults is described in details further in this chapter with explanation of fault masking mechanisms.

Table 9.2: Sorts of undetected faults due to the PicoBlaze blocks

Sort of undetected fault / Processor block	Additional component of logic sum	Additional variables of sum components	Reduced logic sum component	Reduced variable of one of logic product	Changed Variables of components of logic sums	Hardware redundancy
2.Interrupts logic	20 /38	8/38	2/38	8/38	0/38	0/38
3. Decoder for control PC, CALL /RETURN STACK	14/37	9/37	2/37	6/37	6/37	0/37
4. ZERO and CARRY flags	7/32	4/32	6/32	6/32	9/32	0/32
5. Program Counter	1/11	3/11	1/11	4/11	2/11	0/11
8.Logical operations	1/1	0/1	0/1	0/1	0/1	0/1
9. Shift and Rotate operations	3/14	1/14	3/14	4/14	3/14	0/14
10.Arithmetical operations	1/4	1/4	1/4	1/4	0/4	0/4
11.ALU multiplexer	16/84	29/84	5/84	13/84	21/84	27/84
12. Read and Write Strobes	29/43	0/43	0/43	13/43	1/43	0/43
13.CALL/RETURN stack	4/22	2/22	7/22	7/22	2/22	0/22
SUM	96/286	57/286	27/286	62/286	44/286	27/286

Example for processor block: 8 Logical operations. Type of fault 1. Additional component of logic sum. Fault no. 1054: **X"6E8A" => X"6E8B"** remains undetected. This signal is utilized to form logical_result signal. Original VHDL description of this sub-block is presented in Figure 9.3a. Its description after change of the „INIT" parameter is shown in Figure 9.3b.

```
logical_LUT3: LUT4
  generic map (INIT => X"6E8A")
 port map(  I0 => second_operand(7),
            I1 => sx(7),
            I2 => instruction(13),
            I3 => instruction(14),
             O => logical_value(7));
```

(a)

```
logical_LUT3: LUT4
  generic map (INIT => X"6E8B")
 port map(  I0 => second_operand(7),
            I1 => sx(7),
            I2 => instruction(13),
            I3 => instruction(14),
             O => logical_value(7));
```

(b)

Figure 9.3: VHDL implementation of logical LUT a) correct parameter b) modified parameter

Above logical functions for original and changed „INIT'' parameters were derived using methods of Karnaugh maps, Thus implementation on the base of correct VHDL description is following:
O = (I0 I1 (!I3)) + ((!I0) I1 I3) + (I0 (!I1) I3) + (I0 (!I2)).

After above change of parameter:

O = (I0 I1 (!I3)) + ((!I0) I1 I3) + (I0 (!I1) I3) + (I0 (!I2)) + ((!I1) (!I2) (!I3)). Such a parameter modification brought „**additional component of logic sum"**: ((!I1) (!I2) (!I3)). In order to discover it, bit 7^{th} of register sx has to have value „0''. Moreover bits 13^{th} and 14^{th} of a logical instruction has to have value „0". This is satisfied exclusively by „LOAD" logical instruction. **Discovery of this fault is possible**, but requires additional manipulation. However such an operation as zeroing of every register to ensure the correctness of initial value is executed for every register, and often more times upon any registers, but then every register is filled many times with different data. In this way this fault is masked. In order to detect this fault. I would have to execute LOAD sx, 00, and then for instance XORing result of execution LOAD with register containing final data. In this way one can observe if after execution LOAD sx, 00, there appears 0x10

in the sx register. There is only one undetected fault of this sort, and its detection mechanism was not implemented next to the bijective program supported by LFSR solutions.

Example for processor block: 10. Arithmetical operations. Type of fault: 2. Additional variables of sum components. Fault no.1257: **X"1F" => X"1D"** remain sum detected. This function detects arithmetical operations and passes theirs result, when logical value of the function is equal „0''. Original and modified VHDL description of this sub-block is presented in Figures 9.4a and 9.4b respectively.

```
sel_arith_LUT: LUT3
generic map (INIT => X"1F")
port map( I0 => instruction(14),
      I1 => instruction(15),
      I2 => instruction(16),
      O => sel_arith );
```

a)

```
sel_arith_LUT: LUT3
generic map (INIT => X"1D")
port map( I0 => instruction(14),
      I1 => instruction(15),
      I2 => instruction(16),
      O => sel_arith );
```

b)

Figure 9.4: VHDL implementation of arithmetical operation detection LUT
a) correct parameter b) modified parameter

Logical equation of the correct LUT function obtained in the same way as in previous example is as present:

O = ((!I0) (!I1)) + (!I2).

While modified of LUT parameter generates logic function as follows:
O = ((!I0) (!I1)) + (!I2) I1. It can be seen, that the second component of the logic sum depends additionally from I1. To the LUT input I1 is applied bit 15^{th} of the assembler instruction. Looking to the table „PicoBlaze instructions codes'' (PicoBlaze User Guide attached on CD), it can be deducted, that this fault cannot block flow of results generated by arithmetic instructions. Arithmetic instructions as ADD, ADDCY, SUB, SUBCY, which generates outcomes has 16^{th} bit of theirs assembler instruction equal „1''. Thus component of the LUT logical equation: (!I2) I1, resulting from changing of parameter „INIT'', takes every time logical value „0'' independently from I1 – 15^{th} bit of assembler instruction. Results from other instructions are not passed by this part of hardware (XILINX FDR flip flop). Thus this error cannot disturb theirs data flow too. So this fault is

undetectable by any assembler code, but harmless too, in terms of applications executed on this processor core. This is an instance of fault categorized as „**Additional variables of sum components" (logic redundancy)**.

Example for processor block: 4. ZERO and CARRY flags. Type of fault: 4. Reduced variable of one of logic product. Other undetected fault number 154 **X"41FC" =>X"41FD"** concerns forming of flag_enable signal. This signal drives Clock Enable (CE) input of the Xilinx FDRE flip-flop, and in order to enable of propagation of proper CARRY_FLAG this signal should take value „1''. Original and modified VHDL description of this sub-block is presented in Figures 9.5a and 9.5b respectively.

```
flag_type_LUT: LUT4
   generic map (INIT => X"41FC")
 port map(  I0 => instruction(14),
     I1 => instruction(15),
     I2 => instruction(16),
     I3 => instruction(17),
     O => flag_type);
```

(a)

```
flag_type_LUT: LUT4
   generic map (INIT => X"41FD")
 port map(  I0 => instruction(14),
     I1 => instruction(15),
     I2 => instruction(16),
     I3 => instruction(17),
     O => flag_type);
```

(b)

Figure 9.5: VHDL implementation of flag enable LUT
a) correct parameter b) modified parameter

Equation derived from original LUT function is as follows:
O = (I2 (!I3)) + ((!I0) (!I1) (!I2) I3) + ((!I0) I1 I2) + (I1 (!I3))

After modification of INIT parameter, LUT generates following function:
O = (I2 (!I3)) + ((!I0) (!I1) (!I2)) + ((!I0) I1 I2) + (I1 (!I3)).

Here visible is lack of I3 in second component of this logic sum („**Reduced variable of one of logic product"**). If we look at the PicoBlaze user manual, we can notice, that at lack of I3 (bit 17th) of PicoBlaze instruction, „LOAD'' instruction can set to „1'' the signal flag_type and consequently can set the signal flag_enable. However the LOAD instruction does not generate any flag, thus generation of flag_type signal by this instruction has no effect. Thus result of setting

the flag_type signal by LOAD cannot disturb data flow. This fault is harmless and undetectable by any assembler code.

However lack of I3 has influence on operation of other instructions, we can notice on base of PicoBlaze User Guide, that component ((!I0) (!I1) (!I2) I3)) responses for setting flag_type signal of „SHIFT'' and „ROTATE" instructions. However all these instructions will set flag_type signal correctly despite lack of I3 (17^{th} bit of assembler instruction).

Example for processor block: 4. ZERO and CARRY flags. Type of fault: 5. Changed variables of components of logic sums. Integral part of **the odd parity generation** block at **TEST** instruction is a LUT which realizes select parity function. This function drives Xilinx multiplexer MUXCY which selects „Parity" bit or Select CARRY bit. There are 14 undetected faults concerning this LUT. All these faults are undetected due to the fact, that logical value after change of INIT parameter remains the same despite the fact derived equation is different. In Figure 9.6 original and modified LUT functions are presented.

```
sel_parity_LUT: LUT4
   generic map (INIT => X"F3FF")
port map(  I0 => parity,
       I1 => instruction(13),
       I2 => instruction(15),
       I3 => instruction(16),
       O => flag_type);
```

(a)

```
sel_parity_LUT: LUT4
generic map (INIT => X"F3FE")
port map(  I0 => parity,
       I1 => instruction(13),
       I2 => instruction(15),
       I3 => instruction(16),
       O => flag_type);
```

(b)

Figure 9.6: VHDL implementation of select parity LUT
a) correct parameter b) modified parameter

Correct LUT function is INIT => X"F3FF":

O = !I1 + I2 + !I3

After INIT parameter change, logical function implemented by LUT is as follows: O = (!I0 !I3) + I2 + (!I1 + I3) + (I0 !I3). **„Changed variables of components of logic sums"** When we substitute proper bits of assembly instruction as I1, I2, I3 for „TEST" instruction, it results that logical value of this modified function is 0, as an original one. Values of I1 = 13^{th} bit of the „TEST" instruction every time is equal „1", I2 = 15^{th} bit of instruction is equal „0", I3 = 16^{th} bit of instruction is equal „1". Exactly the same situation is after parameter INIT changes:

INIT => X"F3FD": Modified LUT function has the following form:

O = (I1 !I3) + I2 + (!I1 I3) + (!I0 !I3).

Other examples:

INIT => X"F3FB": O = (I0 !I3) + I2 + !I1,

INIT => X"F3FB": O = (I0 !I3) + I2 + !I1

INIT => X"EF": O = (I0 !I3) + (I1 + !I3) + (!I1 I3) + (I2 I3) + (!I2 !I3).

Situation is the same for others 9 parameter changes.

Example for processor block: 9. SHIFT and ROTATE operations. Type of fault: 3. Reduced logic sum component. A few problems with fault detectability appear inside **shift block**. For instance Look Up Tables of function realizing multiplexer for shifts: shift_mux_LUT0 is presented in Figure 9.7:

```
shift_mux_LUT0: LUT3
generic map (INIT => X"E4")
port map( I0 => instruction(3),
      I1 => shift_in,
      I2 => sx(1),
      O => shift_value(0));
```

(a)

```
shift_mux_LUT0: LUT3
generic map (INIT => X"A0")
port map( I0 => instruction(3),
      I1 => shift_in,
      I2 => sx(1),
      O => shift_value(0));
```

(b)

Figure 9.7: VHDL implementation of shift multiplexer LUT

a) correct parameter b) modified parameter

Equation derived from original LUT function is as present:

O = (I0 I2) + (!I0 I1)

After modification of INIT parameter, LUT generates following function:

O = I0 I2, where component (!I0 I1) disappeared.

It is possible to investigate on the basis of PicoBlaze user manual, that right shifts, which have bit 3rd of instruction equal ,,0" will generate correct results. This fault will be masked if we execute right

shifts. Whereas left shifts, which have bit 3rd of instruction equal „0”, with proper shift_in value equal „0” can discover the error. The shift_in value is calculated by other logical function LUT-implemented.

Thus, on the basis of above arguments, it is visible, that mechanism of detection of fault injected inside „SHIFT” block is multi-steps. This means, that data is passed through more circuits. These circuits are generated by more logical functions. In turns, operation of logical functions may depend on both from choice of an assembler instruction and often quite complex data composition.

Example for processor block: 11. ALU multiplexer. **Type of fault: 6. Hardware redundancy.** Other sort of possible problems related to fault detectability are caused by hardware redundancy and „Pipelining”. Before ALU multiplexer is built in „OR” functor, which collects information incoming from blocks implementing shifts, logical and arithmetical operations. This is a simplified solution which works in this way, that only one input of the OR gate can be driven by „1” in given period of time, whereas other inputs, with results from other blocks should be driven by „0”. The problem of data conflict may occur for pipeline architecture of this simplified ALU. This problem consist in generation of faulty „1” instead „0” by one of the two other faulty functions. This error is hard to detected because of simplified ALU (OR gate for 3 types of operations arithmetic, logical and shifts). There exist some instructions which can open concurrently Xilinx FDR flips flops by instruction detection mechanism. In this way faulty result „1” from shift instructions can be masked by results of execution such arithmetical instructions as ADD, ADDCY… The input from individual blocks is realized by the following function presented in Figure 9.8:

```
or_LUT0: LUT3
generic map (INIT => X"FE")
port map( I0 =>logical_result(0),
          I1 => arith_result(0),
          I2 => shift_result(0),
          O => alu_group(0) );
```

Figure 9.8: VHDL implementation of „OR” LUT

The described above data conflict can mask not only wrong results from arithmetical, logical and shift instructions, but detection of faults injected in the look-up table realized function of „OR" LUT may be harder. There remained 27 undetected faults by the bijective program without LFSR solution. These faults consist in change of the INIT parameter of exactly this function presented in Figure 9.9. Function of Or_LUT is described by Karnaugh map in Figure 9.9:

I1/I2 **I0**	**00**	**01**	**11**	**10**
0	0	1	1	1
1	1	1	1	1

Figure 9.9: Karnaugh map for implementation of „OR" LUT function

I0 = logical result , I1 = arithmetical result, I2 = shift_result

O = I1 + I2 + I0

The hardware realizing the described above block is presented in Figure 9.10:

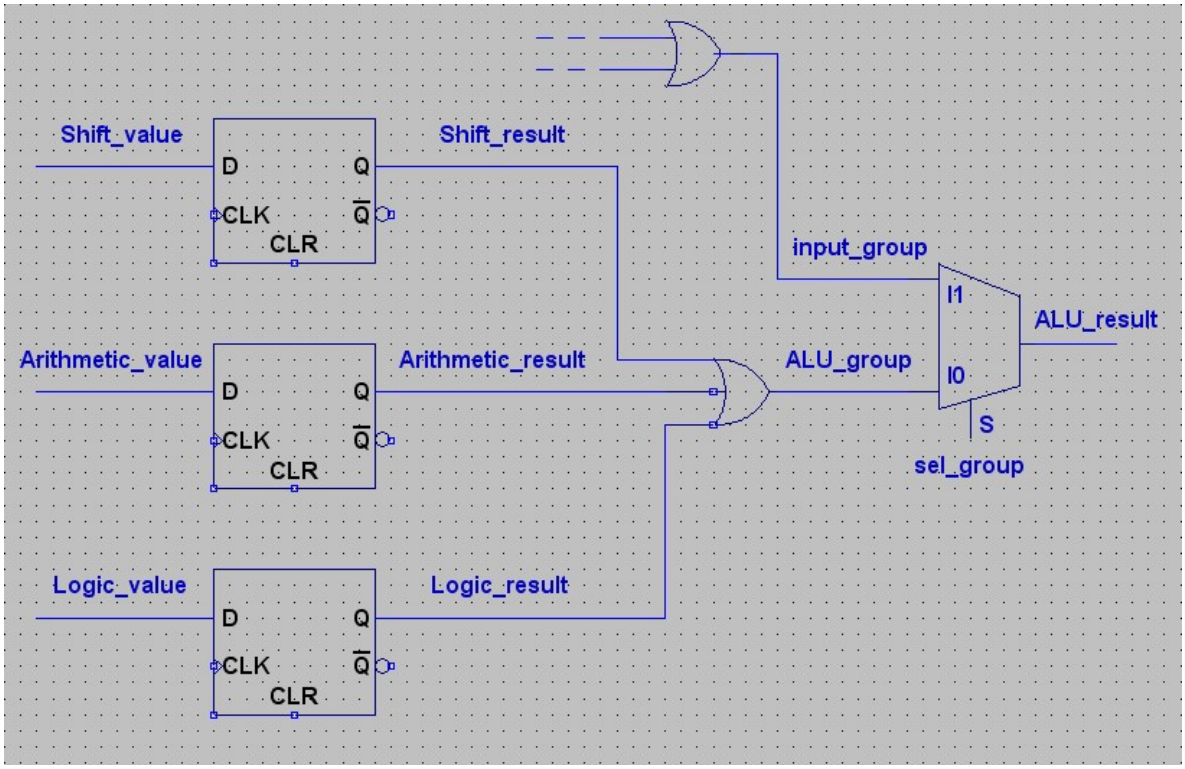


Figure 9.10: The OR functor – the hardware before ALU multiplexer

Presented circuit is responsible exclusively for generation of one bit of vector. Then the same hardware is generated more times by ,,for" loop in vhdl.

Bijective program to testing „SHIFTS", particularly SRA instruction is executed inside the block of instructions which can fulfill bijective property. SRA instruction is executed once inside this block. Next is executed ADDCY (ADD with carry instruction). So faulty result of execution SRA is present by too short period to could be detected. Detector of instructions with its state machine masks this wrong ,,1".

For instance, simulations of the bijective block with SRA instruction showed that results were the same for fault free processor and processor with injected faults „E5" and „F5".

However an experiment where „SHIFT" instructions were executed more times one by one discovered, that faults ,,E5" and „F5" have been detected. The situations are visible on the attached simulation waives to this chapter on CD. Slide E5E4F5_bij.jpg shows that there is no difference on the out_port in proper time, when ,,SHIFT" instruction was executed only once. Here value of parameter INIT was changed first to ,,E5" and then to ,,F5".

Slide E5E4F5_3SRA.jpg shows the difference on the out_port in case, where „SHIFT" instructions were executed more than once. Here value of parameter INIT was changed first to „E5" and then to „F5" as before. Previous block to SRA testing looks as in Listing 9.1 below:

```
      LOAD    s0,  00;
      LOAD    sC,  00;
      LOAD    sF,  00;
      JUMP    ld;
dod:  ADD     s0,  01;
 ld:  LOAD    sC,  s0;
      TEST    sC , 01;
      SRA     sC
      ADDCY   sF,  00;
      RR      sF
      OR      sC,  sF;
      AND     sF,  sC;
      AND     sC,  7F;
      OR      sC,  sF;
      LOAD    sF,  00;
      OUTPUT  sC,  ff;
      JUMP    dod;
```

Listing 9.1: Previous block to SRA testing

```
      LOAD    s0,  00;
      LOAD    sC,  00;
      LOAD    sF,  00;
      JUMP    ld;
dod:  ADD     s0,  01;
ld:   LOAD    sC,  s0;
      TEST    sC , 01;
      SRA     sC;
      RR      sC;       signal from Shifts is present by more clocks.
      ADDCY   sF,  00;
      RR      sF;
      OR      sC,  SF;
      AND     sF,  sC;
      AND     sC,  7F;
      OR      sC,  sF;
      LOAD    sF,  00;
      OUTPUT  sC,  FF;
      JUMP    dod;
```

Listing 9.2: Modified block which detected „E5" and „F5" faults

There is no SRA instruction preceding ADD or ADDCY instruction. So, faults „E5" and „F5" have been detected. And here a problem appears; how to implement bijective block to SRA testing which does not obscure results.

The bijective block based on LFSR solution to „SHIFTS" testing detected all faults injected into hardware realizing or_LUT0 up to or_LUT7. Construction of LFSR blocks is such, that do not occur arithmetical or logical instruction, which can change data in the direct neighbourhood of any shift instruction. Examples of code snippets are presented and described below. First one example in Listing 9.3 concerns LSFR block to „Shift Right Arithmetical" (SRA) test.

```
.......
TEST    s8, 01;
LOAD    s7, 00;
SRA     sD;
TEST    sD, 01;
ADDCY   s9, 00;
.....................
```

Listing 9.3: Code snippet of program LFSR based to „Shift Right Arithmetical (SRA) test

In this example before and after „SRA" are executed instructions which cannot generate any „1" as „ALU result". „TEST" doesn't change content of any register. „LOAD" into s7 „0" cannot generate any „1" result. „ADDCY" is not executed in immediate vicinity of the „SRA" instruction. Next example in the Listing 9.4 is intended to SR0 test:

```
.................
TEST    s8, 01;
ADDCY   s7, 00;
SR0     sD;
RR      s7;
OR      sD, s7;
.............
```

Listing 9.4: Code snippet of program LFSR based to „Shift Right „0" fill" (SR0) test

Here the situation is similar; „ADDCY" cannot generate as result any „1". „RR" (Rotate Right) belongs to the same block of instruction as SR0 (Shift right „0" fill).

Example to „SR1" testing (bellow) is slightly different (see Listing 9.5). Only visible difference is, that directly after „SR1" (Shift Right „1" fill) executed is logical instruction „AND". This „AND" is able to modify exclusively one bit (erase bit 7th). There is no way to get a wrong result as „1" on active bits with data since 6th down to 0.

```
TEST     s8, 01;
ADDCY, s7,  00;
SR1      sD;
AND      sD, 7F;
RR       s7;
```

Listing 9.5: Code snippet of program LFSR based to „Shift Right „1" fill" (SR1) test

There were at least about 27 undetected faults related to „Or_LUT" functions at previous test programs.

I have carried out detailed analysis of about 30% of undetected faults (about 100 faults) of all categories collected in Table 9.2. I had opportunity to observe, that every time in case of undetected faults, selected randomly, merely a few the same schemes of fault masking as described at beginning of this chapter are repeated. Thus conclude, that above mentioned categories of redundancies make impossible detection of remained faults. On the other hand, it can be considered as a positive feature of a processor, that certain SEU induced faults cannot disturb its operation. The effort of working on the analyses turned out to be fruitful. These analyses proved, that it is impossible to achieve 100% of fault coverage of injected faults for PicoBlaze and similar processor cores implemented in FPGAs, where SEU-induced faults are modeled in the same way.

10. Conclusions

The proposed approach of testing processor cores implemented in FPGAs produces compact test sequences, that detects permanent SEU-induced faults of embedded processor cores implemented in SRAM-based FPGAs. The notion of the sensitive path from the automatic test-pattern-generation (ATPG) techniques proposed in **(Doumar 1999)** has slightly different meaning. My novel assumption is, that the test sequence represents a sensitive path, if the data flow through it is sensitive to changes of the input pattern **(Wegrzyn 2009)**.

On the basis of novel assumption and test methodology two theses were formulated. The first thesis is:

Using the sensitive path principle which employs the bijective property of test program may considerable simplify testing procedure and improve fault coverage.

One of the most important novel idea of this thesis is the proposed bijective property of the test program. Further, I have proposed methods of assurance of bijectivity on assembler instructions level, as described in chapter 5, and next I have proposed assembler test program implemented accordance to these principles. Program composed from bijective blocks achieve significantly better fault coverage (85,6%) than well-known computing application or test programs with simpler architectures achieved. Definitively, the best fault coverage (94,76 %) I have achieved creating local bijective test programs, with generating simultaneously complete cycle of local test vectors. It should be noticed that SUE-induced faults are more difficult to detect in comparison with the stuck-at faults which are mostly referred in literature.

The cyclic usage of test vectors as new input tests can be utilized at industrial testing, where it is enough to store only one (start) vector and the number of iterations in reference memory. The cyclic usage of test vectors is suitable for BIST, particularly where FPGAs work under difficult conditions i.e. are exposed to increased radiation.

The second thesis is:

Optimization heuristics combined with the proposed fault injection methodology can significantly reduce the number of test vectors required to achieve maximal fault coverage of soft-processors implemented in FPGAs.

One of the most important novelty introduced hereby is a novel model of injected faults. The functional model of faults differs considerably from the conventional stuck-at fault model due to the fact that SEU-induced faults affect FPGA configuration SRAM and thus logic implemented by look-up tables (LUT). Thus testing and fault model in FPGA differs substantially from well-known stuck-at model in ASICs. No one before, had introduced such a fault model based on proposed principles **(Wegrzyn 2009)**, **(Wegrzyn 2014 A)**. I have reflected SEUs induced faults in LUTs accordance to nature of these physical phenomena in semiconductors and range of their appearance as described in bibliography of subject (**Gaspard 2017**), **(Rebaudengo 2002 A), (Rebaudengo 2002 B)**. Benefits of my novelty are double, because in this way it is possible to model SEU faults in LUTs, which lead to different implementations of logical functions as these intended. Second benefit is, that these faults can be interpreted in particular cases as a stuck-at „0" or stuck-at „1" faults at inputs or output of LUTs, so the FPGA routing resources are also simulated. This together with optimization heuristics significantly reduces the number of test vectors required to achieve maximal fault coverage.

I have developed my original test environment with Perl script driving CADENCE NC VHDL tool. An evaluation case study of the functional test for the PicoBlaze processor core was performed on a generalized fault model, including both stuck-at faults and functional faults in LUTs, which are more difficult to detect. The achieved fault coverage confirms the efficiency of the proposed approach. Despite the fact that SEU induced faults in LUTs are harder to detect, I have obtained results comparable to other publications, which utilized only stuck-at fault model.

Three strategies to minimize the number of test patterns dedicated for bijective, but not fully cyclic program are proposed by the author in this dissertation. First Algorithm 1, so called **„Greedy"**, requires 33 vectors to obtain maximal Fault Coverage. Algorithm 2 so called **„The lowest order vectors first"**, results in 28 test vectors, instead of all possible 256 vectors in case of PicoBlaze. Algorithm 3 - **„Hybrid"** is a mixture of Algorithm 2 and 1. The best results has been achieved by the Hybrid Algorithm. Algorithm 3 is especially useful in the case when the number of the lowest-

order faults is equal or larger than two, as in this case there are two or more vectors that cover the same fault.

I have checked how many vectors are required for testing individual processor blocks too. The most difficult to testing block turned out to be the block which generates ZERO and CARRY flags. The conclusion can be driven from the experiments described in chapter 6.2.4 that for most hardware blocks two local vectors are enough to obtain FCmax. The exception is arithmetical block (three local vectors) and flags module which requires 28 vectors. Consequently, in the case when the test run time is crucial, local rather than global vectors should be used.

The next important issue is execution time of the complete test sequence. Initially the maximal fault coverage had achieved by usage only 28 global test vectors for earlier version of bijective program. In this case execution time of these 370 instructions 28 times is estimated at 400µs. For comparison, testing of the PicoBlaze with a method of configuration read back could take about 3,5 s at the same equipment (however in general case faster read back interfaces are available). My newest researches have revealed, that it is enough to apply exclusively one test vector to achieve the full fault coverage for some processor blocks, if large percent of the sensitivity paths is activated. This has been assured by the last version of fully bijective test program supported by LFSR solutions with local test vectors.

The achieved results show, that developed methodology of fully bijective test program with fully active sensitivity paths is suitable for use for testing of the microprocessor blocks executing logical, arithmetical and shift operations. Hereby I presented the best achievements. It does not detect only 1,2% faults in these blocks in total. These faults may not be detectable regardless of the any test program, due to a few kinds of logical or hardware redundancies as I have analyzed fault masking in chapter 9. While these methodology is not especially dedicated to exhaustive testing such blocks of microprocessor as I/O, Interrupt controller, Program Counter. However testing of these block with my testing program can bring good fault coverage, but at a relatively large amount of work and time.

All assumptions formulated in theses were proved, and the assumptions and goals of the work were realized. Moreover author's researches have proved, that achievement of 100% FC is impossible by any test program written in assembler for reason of logical and hardware redundancies in case of fault injected in logical functions realized by LUTs for more complex systems implemented in FPGAs.

Benefits of the novel model for fault injection:

- Reflects SEU induced faults in LUTs accordance to the nature of these physical phenomena in semiconductors as described in bibliography.
- Fault modeling based on modification of logical functions implemented by LUTs.
- Fault modeling of stuck-at faults on inputs and outputs of LUTs (as interpretation of faults in LUTs).
- Faults injected only on one bit position in LUT memory – this reflects correctly real SEUs and constitutes bigger challenge for test program.
- Faithful fault model combined with automated simulation of every possible fault allows to limit considerably the number of test vectors and test time.
- This fault model is suitable for SRAM based FPGA families, moreover fault model is different as for ASIC (usually stuck-at faults and stuck between wires).
- Fault modeling in LUTs is feasible from FPGA primitives library level.

Original achievements of the author:

- Introduction of the novel SEU-induced fault model dedicated to SRAM-based FPGAs.
- Elaboration of complete system for testing of soft processor cores implemented in FPGA.
- Implemented Pearl script which performs automated fault injection accordance to this novel model and controls CADENCE simulator.
- Introduction of novel bijective testing methodology based on data sensitive path principle.
- Significantly better efficiency achieved by bijective test program than other computing applications e.g. matrix multiplication.
- Development of original bijective methods on assembler instructions level.
- Compact size of bijective program about 370 assembler instructions.
- Detection, investigation and solution of the problem of lack of full cycle.
- Achieved definitively better fault coverage by program composed from local subprograms, which generate completely cyclic results than other applications.

- Activation of high percent of data sensitive paths by bijective completely cyclic test program.
- Reduced set of test to only one vector to achieve FCmax for the test program composed on LFSR principle (completely bijective).
- Development of efficient cyclic methodology for industrial testing (only start test vector and number of iterations stored in reference memory).
- Cyclic methodology useful for BIST too, when FPGA is explored on radiation.

Proposed novel optimization heuristics:

- Nine times reduced number of required global test vectors, what reduces considerably test time.
- Execution the test program can be many times faster than read back in some cases.
- Individual processor block requires on average only 2 local vectors to achieve FCmax.
- FC over 95% FCmax at first 3-4 global test vectors.

Additional conclusions from research works:

- Bijective property without assurance of full cycle is insufficient to achieve FCmax.
- Achievement of 100% FC is impossible by any test program written in assembler for reason of logical and hardware redundancies.
- Possible slight improvement of FC by detailed analyses of interactions of input data and processor HW.
- The hardest to test is block which generates ZERO and CARRY flags.

Future works:

- Testing individual blocks of bigger processor cores.
- Optimization and automation of generation of bijective blocks of test program.
- Development of fault injection simulators environment based on other commercial tools. Not only CADENCE based.
- Testing of interrupt and Floating Point Unit blocks.

11. Appendix

11.1. Dissertation - Electronic version of the dissertation

The dissertation was written using Microsoft Office 365. The dissertation is divided into then chapters and appendix. The folder „Dissertation" contains PDF version of the work.

11.2. PB - PicoBlaze structural VHDL

The directory „PB" contains original and with unrolled „for" loops code of PicoBlaze processor core in structural VHDL.

11.3. PBUM - PicoBlaze User Manual

The directory „PBUM" contains PicoBlaze User Manual in PDF.

11.4. MBUM - MicroBlaze User Manual

The directory „MBUM" contains MicroBlaze User Manual in PDF.

11.5. ModelSim waves for chapter 9 – hardware redundancy.

The directory „WavesCh9" contains ModelSim waves which illustrate the hardware redundancy.

References:

Abramovici, M., Stroud, C., Hamilton C., Wijesuriya, S., Verma, V. (1999). *„Using Roving STARs for On-Line Testing and Diagnosis of FPGAs in Fault-Tolerant Applications."* Proceedings International Test Conference. IEEE Cat. No. 99CH37034, 973-982.

Abramovici, M., Stroud, C. (2000). „*BIST-Based Detection and Diagnosis of Multiple Faults in FPGAs."* Proceedings International Test Conference(ITC'00), 785-794.

Abramovici, M. Stroud, C. (2002).*„BIST-based delay fault testing in FPGAs."*In Proceedings of the 8th IEEE International On-Line Testing Workshop, pp. 131–134.

Alderighi, M., D´Angelo, S., Mancini, M., Sechi, G.R. (2003 A).*„A Fault Injection Tool for SRAM-based FPGAs."* Proceedings of the 9th IEEE International On-Line Testing Symposium ('03).

Alderighi, M., Casini, F., D´Angelo, S., Mancini, M., Marmo, A., Pastore, S., Sechi, G.R. (2003 B). „*A Tool for Injecting SEU-like Faults into the Configuration Control Mechanismof Xilinx Virtex FPGAs."*18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 71-78.

Batcher, K., Papachristu, C. (1999). „*Instruction Randomization Self Test for Processor Cores."* Proceeding IEEE VLSI Test Symposium, 34-40.

Bernardi, P., Rebaudengo, M., Sonza, M. (2004).*„Using Infrastructure IPs to support SW-based Self Test of Processor Cores."*Proceedings of the 5th International Workshop Microprocessor Test and Verification, 22-27.

Bernardi P., et al.(2008). *„An Effective Technique for the AutomaticGeneration of Diagnosis-Oriented Programs forProcessor Cores,"*IEEE Trans. Computer-Aided Designof Integrated Circuits and Systems,vol. 27, no. 3, pp. 570-574.

Bernardi, P., et al. (2012). „*On-line software-based self-test of the address calculation unit in RISC Processors.*" Proc. 17th IEEE Eur. Test Symposium (ETS), 1–6.

Bernardi, P., Ciganda, L., Sanchez, E., Sonza, M. (2014). „*MIHST: A Hardware Technique for Embedded Microprocessor Functional On-Line Self-Test "*IEEE Transactions on Computers, Vol. 63, no. 11, November 2014, 2760-2772.

Bolchini, C., Miele, A., Sandionigi, C. (2007). *„TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGA."*IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 87–95.

Bushnell, M., Agrawal, V. (2000). „*Essentials of Electronic Testing.*" Kluwer academic publishers, ISBN : 0-7923-799, 1-8.

Campbell, B., Stark, I. (2014). *,,Randomized testing of a microprocessor model using SMT-solver state generation."*FMICS 2014: Formal Methods for Industrial Critical Systems, 185-199.

Chen, L., Dey, S. (2000). *,,A Deterministic Functional Self-Test Methodology for Processors."* Proceeding IEEE VLSI Test Symposium, 255-262.

Chen, L., Dey, S. (March 2001). *,,Software-Based Self-Testing Methodology for Processor Cores."* IEEE Transaction Computer-Aided Design, 369-380.

Chen et al.(2003). *,,A Scalable Software-Based Self-Test Methodology for Programmable Processors."* Proc. 40th Design Automation Conf., ACM Press, pp. 548-553.

Chen, Ch., Wei, Ch., Tai-Hua, Lu. (2007). ,,*Software-Based Self-Testing With Multiple-Level Abstractions for Soft Processor Cores*." IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 15, NO. 5, May 2007, 505- 517.

Civera, P., Macchiarulo, L., Rebaudengo, M., SonzaReorda, M., Violante M. (2001).*,,FPGA-Based Fault Injection Techniques for Fast Evaluation of Fault Tolerance in VLSI Circuits."* FPL 2001, 11th International Conference on Field Programmable Logic and Applications, Belfast (UK), 493-502.

Corno, F., Cumani, G., Sonza, M., Squillero, G. (2002). *,,Evolutionary Test Program Induction for Microprocessor Design Verification."*Proceedings of the 11th IEEE Asian Test Symposium, Guam (USA), 368-373.

Corno, F., Cumani, G., SonzaReorda, M., Squillero, G. (2003). *,,Fully Automatic Test Program Generation for Microprocessor Cores."*Proceedings Design Automation & Test in Europe, 1006-1011.

Corno, F., Sanchez, E., Sonza, M., Squillero, G. (2004). *,,Automatic Test Program Generation: A Case Study."* IEEE Design & Test, Special issue on Functional Verification and Testbench Generation, **21:** 102-109.

Doumar, A., Ito, H. (1999). *,,Testing the logic cells and interconnects resources for FPGAs."* 8th Asian Test Symposium, 369-374.

Dutton, B., Stroud C., (2009). ,,*Soft core embedded processor based built-in self-test of FPGAs,*" 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems.

Gaspard, N.,J., (2017). *,,Single-Event Upset Technology Scaling Trends of Unhardened and Hardened Flip-Flops in Bulk CMOS."*Dissertation for the degree DOCTOR of PHILOSOPHY. Vanderbilt University. Nashville, Tennessee, USA.

Gizopoulos, D., Paschalis, A., Zorian, Y., Psarakis, M. (1997). ,,*An Effective BIST Scheme for Arithmetic Logic Units.*" Proc. Int'l Test Conference, 868-877.

Gizopoulos, D., Paschalis, A., Zorian, Y. (2004). *,,Embedded Processor-BasedSelf-Test"*, V.D. Agrawal, ed. Kluwer Academic Publ,ishers.

Gizopoulos, D., et al. (2008). *,,Systematic Software-Based Self-Test for Pipelined Processors,*'' IEEE Trans. Very LargeScale Integration (VLSI) Systems, vol. 16, no. 11, pp. 1441-1453.

Gurumurthy, S. Vasudevan, S., Abraham, J., A. (2006). *„Automatic Generation of Instruction Sequences Targeting Hard-to-Detect Structural Faults in a Processor."* Proc. Int'l Test Conf. (ITC 06), vol. 2, IEEE CS Press, pp. 776-784.

Haar, S., Jard, C., Jourdan., G. (2007). „*Testing Input/OutputPartialOrderAutomata*.", Testing of Software and Communicating Systems, 171-185.

Hayes, J. P., Polian, I., Becker B. (2007). *„An Analysis Framework for Transient-Error Tolerance."* 25th IEEE VLSI Test Symmposium, 249-255.

Hellebrand S., Zoelin, Ch., G. (2007). *„Testing and Monitoring Nonsocial Systems- Challenges and Strategies for Advanced Quality Assurance."* MIDEM, BLED, 3-10.

Hlawiczka A. (1997). „*Rejestry Liniowe-Analiza,Synteza i Zastosowania w Testowaniu Układów Cyfrowych."* Zeszyty Naukowe Politechniki Slaskiej. Gliwice.

Holbert, (2006). *„Single event effects."* NASA report

Huang, W., Lombardi, F. (1996). *„An approach to testing programmable/ configurable field programmable gate arrays."* Proceeding 14th IEEE VLSI Test Symposium, 450–455.

Karp, S., Gilbert, B., K. (1993). *„Digital system design in the presence of single event upsets."* IEEE Transaction on Aerospace and Electronic Systems,**29:** 310-316.

Kastensmidt, F., L., Carro, L., Reis, R. (2006). *„ Fault-tolerance techniques for SRAM-based FPGAs."* Frontiers in electronic testing. ISBN: 987-0-387-31068-8.

Katz, K., LaBel, J. J., Wang, B., Cronquist, R., Koga, S., Penzin, S., Swift, G. (1997). *„Radiation effects on current field programmable technologies."* IEEE Transactions on Nuclear Science, **44:** 1945-1956.

Katz, R., Wang, J.J., Reed, R., Kleyner, I., D'Ordine, M., McCollum, J., Cronquist, B., Howard, J. (1999). *„The effects of architecture and process on the hardness of programmableTechnologies."* IEEE Transactions on Nuclear Science, **46:** 1736-1743.

King, M., P., (2014). *„Energetic Electron-Induced Single Event Upsets in Static Random Access Memory."* Dissertation for the degree PhD. Vanderbilt University. Nashville, Tennessee, USA.

Kranitis,N., Paschalis, A., Gizopoulos, D., Zorian,Y. (2002 A). *„Effective Software Self-Test Methodology for Processor Cores."* Proceedings Design Automation & Test in Europe 2002 Paris, 592-597.

Kranitis,N., Gizopoulos,D., Paschalis, A., Zorian, Y. (2002 B). *„Instruction-based self-testing of processor cores."* Proceedings 20th IEEE VLSI Test Symp, 223-228.

Kranitis, N., et al.(2005). *„Software-Based Self-Testing of Embedded Processors,"* IEEE Trans. Computers,vol. 54, no. 4, pp. 461-475.

Krstic, A., et al. (2002). *„Embedded Software-Based Self-Test for Programmable Core-Based Designs,"* IEEE Design &Test, vol. 19, no. 4, pp. 18-27.

Kupferschmid, S., Lewis, M., Schubert, T., Becker, B., (2011). „*Incremental preprocessing methods for use in BMC.* " Formal Methods Syst. Design, vol. 39, no. 2, 1–20.

Lesea, A., *et al.* (2005). „*The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs,"*IEEE Transactions on Materials Reliability, September

Leveugle, R., Ammari, A. (2004). *„Early SEU Fault Injection in Digital, Analog and Mixed Signal Circuits: a Global Flow."* Proceedings Design, Automation and Test in Europe Conference and Exhibition,590-596.

Lindsay, W., Sanchez, E., Sonza, M., Squillero, G. (2004). „*Automatic Test Programs Generation Driven by Internal Performance Counters."* 5th International Workshop on Microprocessor Test and Verification, 8-13.

Lingappan L., and Jha, N. K. (2007). *„Satisfiability-Based AutomaticTest Program Generation and Design for Testabilityfor Microprocessors,"*IEEE Trans. Very LargeScale Integration (VLSI) Systems, vol. 15, no. 5, pp. 518-530.

Maheshwari, A., Koren, I., Burleson W. (2004). *„Accurate Estimation of Soft Error Rate (SER) in VLSI Circuits."* 19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 377-385.

Megan, C., Boutte, A., J. „*A comparison of High-Energy Electron and Cobalt-60 gamma-Ray Radiation Testing."* (2012). NASA research paper.

Michinishi, H.Yokohira, T. Okamoto, T. (1996). *„A Test Methodology for Interconnect Structures of LUT-Based FPGAs."*Proceedings of the 5th Asian Test Symposium, pp. 68–74.

Michinnishi, H., Yokohira, T., Okamoto, T., Inoue, T., Fujiwara, H. (1997). *„Testing for programming circuit of LUT-based FPGAs. "*Proceedings of the 5th IEEE Asian Test Symposium, 68-74.

Mishra, P., Dutt, N. (2003). *„A Methodology for Validation of Microprocessors using Equivalence Checking."*Proceedings of the 4th International Workshop on Microprocessor Test Verification Common Challenges and Solutions, 83-88.

Mishra, P., Dutt, N., Kashai, Y. (2004). *„Functional Verification of Pipelined Processors: A Case Study."*Proceedings of the 5th International Workshop on Microprocessor Test and Verification Common Challenges and Solutions, 79-84.

Ohlsson, M., Dyreklev, P., Johansson, K. (2002). *„Neutron single event upsets in SRAM-BASED FPGAs."* Ericsson Saab Avionics AB, Electromagnetic Technology Division, Sweden.

Pereira, G., Andrade, A., Balen, T., R., Lubaszewski, M., Azaïs, F., Renovell, M. (2005). *„Testing the Interconnect Networks and I/O Resources of Field Programmable Analog Arrays."* Proceedings of the 23rd IEEE VLSI Test Symposium, 389-394.

Psarakis, M., Gizopoulos, D., Sanchez, E., Sonza, M. (2010). „*Microprocessor Software-Based Self-Testing."* IEEE Design and Test of Computers, July 2010

Rebaudengo M., Sonza Reorda, M., Violante, M. (2002 A). „*A new functional fault model for FPGA Application-Oriented testing."* Proceedings of 17th IEEE International Symposium on Volume, 372-380.

Rebaudengo, M., Sonza Reorda, M., Violante, M. (2002 B). *„Simulation-based analysis of SEU effects on SRAM-based FPGAs."*International Conference on Field Programmable Logic and Application, 607- 615.

Renovell, M., Figueras J., Zorian, Y. (1997).*„Test of RAM-Based FPGA: Methodology Application to the Interconnect."* 15th IEEE VLSI Test Symposium, 230-237.

Renovell, M., Portal, J., Figueras, J., Zorian, Y. (1998a). *„Testing the Interconnect of RAM- Based FPGAs."* IEEE Designs and Test of Computers,**15:** 45-50.

Renovell M., Portal J.M., Faure P., Figueras J., Zorian Y. (2000 A). *„Analyzing the Test Generation Problem for an Application-Oriented Test of FPGAs."*IEEE European TestWorkshop, 75-80.

Renovell, M., Figueras, J., Zorian, Y. (2000 C). *„TOF : A Tool for Test Pattern Generation Optimization of an FPGA Application-Oriented Test."*Proceedings of the 9th Asian Test Symposium, 323-328.

Renovell, M., Portal J., M., Faure P. (2001). *„ADiscusion on Test Pattern Generation for FPGA-Implemented Circuits."*Journal of Electronic Testing: Theory and Applications,**17:** 283-290.

Renovell, M., Faure, P., Prinetto P., Zorian, Y. (2002). „*Testing the Unidimensionnal Interconnect Architecture of Symmetrical SRAM - based FPGA*." Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications, 297-301.

Riefert, A., Cantoro, R., Sauer, M., Sonza M., (2016).„*A Flexible Framework for the Automatic Generation of SBST Programs.*"IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

Safi, E., Karimi, Z., Abbaspour, M., Navabi, M. (2003). *„Utilizing Various ADL Facetes for Instruction Level CPU Test."*Proceedings of the Fourth International Workshop Microprocessor, Test and Verification, 38-45.

Sanchez, E., SonzaReorda, M., Tonda, A., (2011). „*On the Functional Test of Branch Prediction Units Based on the Branch History Table architecture.*" VLSI-SoC: Advanced Research for System on Chip, 110-123.

Shen,J.,Abraham, J.A. (1998). *„Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation*."Proceedings International Test Conference, 990-999.

Sonza Reorda, M., Sterpone, L., Violante, M., Portela-Garcia, M., Lopez-Ongil C., Enterena L. (2006). *„Fault Injection-based Reliability Evaluation of SoPCs",* 75-82.

Sosnowski, J. (2005). *„Testowanie i niezawodność systemów komputerowych."*Akademicka Oficyna Wydawnicza EXIT, Warszawa 2005.

Stroud C., Leach, K., Slaughter, T. (2003). „*BIST for Xilinx 4000 and Spartan SeriesFPGAs: ACase Study."*Proceedings of the International Test Conference 2003 (ITC'03), 1258-1267.

Stroud, C., Sunwoo, J., Garimella, S., Harris, J., (2004). „*Built-In Self-Test for System-on-Chip: A Case Study.*" Proc. Int'l Test Conf., 837-846.

Stroud C., Dutton, B. (2009).„*Built-in self-test of programmable input/output tiles in Virtex-5, FPGAs,"*IEEE Southeastern Symposium on System Theory, pp. 235–239.

Suthar, V., Dutt, S. (2006).*„Mixed PLB and Interconnect BIST for FPGAs Without Fault-FreeAssumptions."*Proceedings of the 24th IEEE VLSI Test Symposium, 36-43.

Syam, E., Reddy, S., Chandrasekhar, V., Sashikanth, M., Kamakoti, V. (2005). *„Detecting SEU-caused Routing Errors in SRAM-based FPGAs."* Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on EmbeddedSystems Design, 736-741.

Tahoori, B., M. , McCluskey, E. , J., Renovell, M. , Faure, P. (2004 A). *"A Multi-Configuration Strategy for an Application Dependent Testing of FPGAs."* IEEE VLSI Test Symposium, 154- 170.

Tahoori M., B. (2004 B). *„Application-Specific Bridging Fault Testing of FPGAs."* Journal of Electronic Testing: Theory and Applications, **20:** 279-289.

Tai-Hua, L., Chen, Ch., Lee, K. (2011). *„Effective Hybrid Test Program Development for Software-Based Self-Testing of Pipeline Processor Cores"*. IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 19, NO. 3, March 2011,516-520.

Teng, L., Jianbin Z., Jianguo, R., Jianhua, F., Wang, Y.(2009)*., A novel scheme for application-dependent testing of FPGAs."*

Xilinx company Web page, (www.xilinx.com).

„Device Reliability Report. (2013). " Xilinx Corporation, November, UG116 (v. 9.6).

Wegrzyn, M., Novak, F., Biasizzo, A. (2006). *„Application-oriented testing of FPGA circuits."* 42rd International Conference on Microelectronics Devices and Materials MIDEM, Bled, Slovenia.

Wegrzyn, M., Novak, F., Biasizzo, A., Renovell, M.(2007 A). *„Functional test of processor cores in FPGA-based applications."* Proc.Workshop on Electronic Testing 43 *rd* International Conference on Microelectronics Devices and Materials MIDEM, Bled, 177-181.

Wegrzyn, M., Biasizzo, A., Novak, F. (2007 B). *„Application-oriented testing of embedded processor cores implemented in FPGA circuits."* International Review on Computers and Software,**2:** 666-671.

Wegrzyn, M., Biasizzo, A., Novak, F., Renovell, M., (2009). *„Functional Testing of Processor Cores in FPGA-Based Applications."* Computing and Informatics, Vol.28, 2009, 97-113.

Wegrzyn, M., Sosnowski, J., (2014 A). „*Tracing Fault Effects in FPGA Systems*" INTERNATIONAL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS Vol.60, NO. 1, 103-108.

Wegrzyn, M., Sosnowski, J.,(2014 B). „*Testing schemes for systems based on FPGAs processor cores*" Pomiay, Automatyka, Kontrola Vol. 56, nr 01, 483-485.

Wikipedia 2020 A https://en.wikipedia.org/wiki/Bijection

Wikipedia 2020B, https://en.wikipedia.org/wiki/Cyclic_group

Wen, C., H., Wang, L., C., and Cheng, K., T. (2006)*.„Simulation-Based Functional Test Generation for Embedded Processors.''* IEEE Trans. Computers, vol. 55, no. 11, pp. 1335-1343.

Zhang, Y., Li, H., Li, X. (2013). „Automatic Test Program Generation Using Executing-Trace-Based Constraint Extraction for Embedded Processors. "IEEE Transaction on Very Large Scale Integration (VLSI) Systems, July

Zhou, J., Wunderlich, H. (2006). *„Software-Based Self-Test of Processors under Power Constraints."* Proceedings ofthe conference on Design, automation and test in Europe, 430-435.

Niniejszą pracę doktorską napisałem samodzielnie posługując się jedynie materiałami zamieszczonymi w wykazie referencji.

Mariusz Węgrzyn