ISSN 1688-2784

Universidad de la República
Facultad de Ingeniería

Observation Mechanisms for In-Field Software-Based Self-Test

Tesis presentada a la Facultad de Ingeniería de la Universidad de la República por

Julio Pérez Acle

en cumplimiento parcial de los requerimientos para la obtención del título de Doctor en Ingeniería Eléctrica.

Montevideo
Jueves 14 de marzo de 2019

Observation Mechanisms for In-Field Software-Based Self-Test, Julio Pérez Acle.

ISSN 1688-2784

Esta tesis fue preparada en LaTeX usando la clase iietesis (v1.1).
Contiene un total de 113 páginas.
Compilada el Monday 5 August, 2019.
`http://iie.fing.edu.uy/`

Abstract

When electronic systems are used in safety critical applications, as in the space, avionic, automotive or biomedical areas, it is required to maintain a very low probability of failures due to faults of any kind. Standards and regulations play a significant role, forcing companies to devise and adopt solutions able to achieve predefined targets in terms of dependability. Different techniques can be used to reduce fault occurrence or to minimize the probability that those faults produce critical failures (e.g., by introducing redundancy).

Unfortunately, most of these techniques have a severe impact on the cost of the resulting product and, in some cases, the probability of failures is too large anyway. Hence, a solution commonly used in several scenarios lies on periodically performing a test able to detect the occurrence of any fault before it produces a failure (*in-field test*). This solution is normally based on forcing the processor inside the Device Under Test to execute a properly written test program, which is able to activate possible faults and to make their effects visible in some observable locations. This approach is also called Software-Based Self-Test, or SBST.

If compared with testing in an end of manufacturing scenario, in-field testing has strong limitations in terms of access to the system inputs and outputs because Design for Testability structures and testing equipment are usually not available. As a consequence there are reduced possibilities to activate the faults and to observe their effects.

This reduced observability particularly affects the ability to detect performance faults, i.e. faults that modify the timing but not the final value of computations. This kind of faults are hard to detect by only observing the final content of predefined memory locations, that is the usual test result observation method used in-field.

Initially, the present work was focused on fault tolerance techniques against transient faults induced by ionizing radiation, the so called Single Event Upsets (SEUs). The main contribution of this early stage of the thesis lies in the experimental validation of the feasibility of achieving a safe system by using an architecture that combines task-level redundancy with already available IP cores, thus minimizing the development time. Task execution is replicated and Memory Protection is used to guarantee that any SEU may affect one and only one of the replicas. A proof of concept implementation was developed and validated using fault injection. Results outline the effectiveness of the architecture, and the overhead analysis shows that the proposed architecture is effective in reducing the resource occupation with respect to N-modular redundancy, at an affordable cost in terms of application execution time.

The main part of the thesis is focused on in-field software-based self-test of permanent faults. A set of observation methods exploiting existing or ad-hoc hardware is proposed, aimed at obtaining a better coverage, in particular of performance faults. An extensive quantitative evaluation of the proposed methods is presented, including a comparison with the observation methods traditionally used in end of manufacturing and in-field testing.

Results show that the proposed methods are a good complement to the traditionally used final memory content observation. Moreover, they show that an adequate combination of these complementary methods allows for achieving nearly the same fault coverage achieved when continuously observing all the processor outputs, which is an observation method commonly used for production test but usually not available in-field.

A very interesting by-product of what is described above is a detailed description of how to compute the fault coverage achieved by functional in-field tests using a conventional fault simulator, a tool that is usually applied in an end of manufacturing testing scenario.

Finally, another relevant result in the testing area is a method to detect permanent faults inside the cache coherence logic integrated in each cache controller of a multi-core system, based on the concurrent execution of a test program by the different cores in a coordinated manner. By construction, the method achieves full fault coverage of the static faults in the addressed logic.

Resumen

Cuando se utilizan sistemas electrónicos en aplicaciones críticas como en las áreas biomédica, aeroespacial o automotriz, se requiere mantener una muy baja probabilidad de malfuncionamientos debidos a cualquier tipo de fallas. Los estándares y normas juegan un papel importante, forzando a los desarrolladores a diseñar y adoptar soluciones que sean capaces de alcanzar objetivos predefinidos en cuanto a seguridad y confiabilidad. Pueden utilizarse diferentes técnicas para reducir la ocurrencia de fallas o para minimizar la probabilidad de que esas fallas produzcan malfuncionamientos críticos, por ejemplo a través de la incorporación de redundancia.

Lamentablemente, muchas de esas técnicas afectan en gran medida el costo de los productos y, en algunos casos, la probabilidad de malfuncionamiento sigue siendo demasiado alta. En consecuencia, una solución usada a menudo en varios escenarios consiste en realizar periódicamente un test que sea capaz de detectar la ocurrencia de una falla antes de que esta produzca un mal funcionamiento (*test en campo*). En general, esta solución se basa en forzar a un procesador existente dentro del dispositivo bajo prueba a ejecutar un programa de test que sea capaz de activar las posibles fallas y de hacer que sus efectos sean visibles en puntos observables. A esta metodología también se la llama auto-test basado en software, o en inglés *Software-Based Self-Test* (SBST).

Si se lo compara con un escenario de test de fin de fabricación, el test en campo tiene fuertes limitaciones en términos de posibilidad de acceso a las entradas y salidas del sistema, porque usualmente no se dispone de equipamiento de test ni de la infraestructura de *Design for Testability.* En consecuencia se tiene menos posibilidades de activar las fallas y de observar sus efectos.

Esta observabilidad reducida afecta particularmente la habilidad para detectar fallas de performance, es decir fallas que modifican la temporización pero no el resultado final de los cálculos. Este tipo de fallas es difícil de detectar por la sola observación del contenido final de lugares de memoria, que es el método usual que se utiliza para observar los resultados de un test en campo.

Inicialmente, el presente trabajo estuvo enfocado en técnicas para tolerar fallas transitorias inducidas por radiación ionizante, llamadas en inglés *Single Event Upsets* (SEUs). La principal contribución de esa etapa inicial de la tesis reside en la validación experimental de la viabilidad de obtener un sistema seguro, utilizando una arquitectura que combina redundancia a nivel de tareas con el uso de módulos hardware (*IP cores*) ya disponibles, que minimiza en consecuencia el tiempo de desarrollo. Se replica la ejecución de las tareas y se utiliza protección de memoria para garantizar que un SEU pueda afectar a lo sumo a una sola de las réplicas. Se desarrolló una implementación para prueba de concepto que fue validada mediante inyección de fallas. Los resultados muestran la efectividad de la arquitectura, y el análisis de los recursos utilizados muestra que la arquitectura propuesta es efectiva en reducir la ocupación con respecto a la redundancia modular con N réplicas, a un costo accesible en términos de tiempo de ejecución.

La parte principal de esta tesis se enfoca en el área de auto-test en campo basado en software para la detección de fallas permanentes. Se propone un conjunto de métodos de observación utilizando hardware existente o ad-hoc, con el fin de obtener una mejor cobertura, en particular de las fallas de performance. Se presenta una extensa evaluación cuantitativa de los métodos propuestos, que incluye una comparación con los métodos tradicionalmente utilizados en tests de fin de fabricación y en campo.

Los resultados muestran que los métodos propuestos son un buen complemento del método tradicionalmente usado que consiste en observar el valor final del contenido de memoria. Además muestran que una adecuada combinación de estos métodos complementarios permite alcanzar casi los mismos valores de cobertura de fallas que se obtienen mediante la observación continua de todas las salidas del procesador, método comúnmente usado en tests de fin de fabricación, pero que usualmente no está disponible en campo.

Un subproducto muy interesante de lo arriba expuesto es la descripción detallada del procedimiento para calcular la cobertura de fallas lograda mediante tests funcionales en campo por medio de un simulador de fallas convencional, una herramienta que usualmente se aplica en escenarios de test de fin de fabricación.

Finalmente, otro resultado relevante en el área de test es un método para detectar fallas permanentes dentro de la lógica de coherencia de cache que está integrada en el controlador de cache de cada procesador en un sistema multi procesador. El método está basado en la ejecución de un programa de test en forma coordinada por parte de los diferentes procesadores. Por construcción, el método cubre completamente las fallas de la lógica mencionada.

Contents

Esta página ha sido intencionalmente dejada en blanco.

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

In several domains (e.g., automotive, biomedical, space and aircraft industries) electronic systems are commonly used in mission- and safety-critical applications. In these domains, a misbehavior due to a defect affecting the hardware may have catastrophic effects, including hurting humans and provoking huge economic losses. Hence, there is a strong push to devise techniques able to minimize the probability that a misbehavior caused by a defect arises, and to suitably handle it in case it occurs anyway. When considering the latter point, different solutions have been proposed, and the best solution depends on the specific constraints of each scenario. Standards and regulations (e.g., IEC 61508 for generic safety-related industrial systems [1], ISO 26262 for automotive applications [2], RTCA/DO-254 for avionics [3]) also play a significant role, forcing companies to devise and adopt solutions able to achieve some predefined target in terms of dependability.

Most of the electronic systems involved in safety-critical applications include a microprocessor or a microcontroller. For these systems, it is possible to force programmable units to run test programs able to reveal the presence of defects by activating them and propagating their effects up to an observable location (e.g., a special memory area). Eventually, the application may trigger suitable actions to prevent catastrophic consequences, such as turning the system to a safe status, or reconfiguring it so that the faulty module is not used any more. To minimize the impact on the system, these test programs are often limited to use the time periods left idle by the core applications, or run during the start-up/power-off phases. Such an approach is referred to as Software-Based Self-Test (SBST) [4], and generically labeled as "functional" as it relies directly on the normal functions of the system. SBST does not require any specific Design-for-Testability (DfT) structure, although it may exploit available hardware features, and can be used to test any processor-based system, whether it is a System-on-Chip (SoC) or a board. As a major advantage, testing based on SBST can be run at the processor operational speed, thus allowing the detection of defects which are only activated at the maximum frequency. For this reason, it is often used during the manufacturing test phase as a supplement to other techniques to increase the final defect coverage.

SBST is currently adopted in quite different test scenarios, including both end-of-manufacturing test and in-field test. When applied for end-of-manufacturing test, automatic test equipments (ATE) can either drive the processor inputs while it executes the program and observe the outputs, or load a program into the cache of the processor, force it to execute at full speed, and eventually extract some test syndrome from a special (hidden) register. Additionally, when in-field SBST is considered, a common solution lies in storing the test program in a flash memory, activating its execution when required, and finally checking the content of some selected memory variables, where the test program stores its results.

When comparing the SBST solutions adopted for end-of-manufacturing test with those for in-field test, a major difference is that the former can, in some cases, benefit from full accessibility to the input and output signals of each device (such a test scenario is called “open loop test” in [5]). On the contrary, in solutions oriented to in-field testing, the tester cannot be used and existing DfT structures are in most of the cases not available (e.g., because they have been destroyed or made inaccessible to better protect the system security, or because they are not documented by the device providers). Hence, the only feasible solution for the system company in charge of developing the in-field test is to adopt a purely functional approach, i.e., without resorting to any DfT features. Additionally, in-field constraints may be quite severe: for example, the memory area usable by the test could be limited to a specific size and location, and some faults may become functionally untestable [6] (i.e., no test stimuli exist for them under the in-field test scenario). Although untestable faults by definition cannot affect the system behavior, they may significantly limit the fault coverage that can be achieved, even using a high-quality test program. Therefore, it is desirable to be able to identify untestable faults.

The SBST approach is experiencing an increasing success, mainly because it offers the possibility to the semiconductor company manufacturing the device (and knowing its internal structure) to develop the test code, grade it in terms of achieved fault coverage, and pass it to the system company, which eventually integrates it in the application code. The system company is also in charge of developing the code responsible for launching the test and retrieving the results it produced, managing the situations when a fault is detected.

Since the test code is often activated in small chunks, whose execution can fit in the idle times of the application, it is convenient to organize it in a set of procedures, composing a *Self-Test Library* (STL). STLs are currently offered by several semiconductor and IP companies [7–12].

When developing the code of an STL, special techniques must be followed to activate the target faults and to make them visible. The latter point is particularly important, especially because during in-field test the observability of the DUT behavior is necessarily limited. Hence, several solutions can be adopted, possibly involving the support of existing or ad hoc hardware.

Concerning observability, some solutions adopted for end-of-manufacturing test may allow the continuous monitoring by the ATE of all the output signals of the device under test. On the contrary, with in-field SBST the ATE cannot be used, and thus the effects of faults are typically observed by checking, at the end of the test program execution, the values left by the program in some specified memory locations. This limited observability may significantly reduce the achievable fault coverage; some specific fault categories are known to be untestable if fault detection is only based on looking at the final memory content. In particular, faults that only affect the time behavior of the processor (e.g., by delaying some operation) found in modules such as Cache Controllers [13] and Branch Prediction Units [14] cannot be detected in this way. The test of these performance faults [15] can be successfully faced by resorting to the so-called performance counters existing in most of the current microprocessors and microcontrollers [16]. Alternatively, one can resort to special hardware modules that can be added to a processor, able to monitor the bus during the execution of a test program and then compute a signature. As a result, re-using any test program developed for high-observability end-of-manufacturing SBST for in-field SBST may be either very expensive, or result in a significant drop of the achieved fault coverage. Recently, some papers specifically focused on the generation of test programs for in-field SBST [17].

Taking into account the panorama presented above, some of the questions that motivated the work on the present thesis are:

- How to enhance the fault coverage achieved by in-field software-based self-tests?
- When developing an in-field test, is it possible to reuse the design effort used to develop a test program for an end of manufacturing scenario? How can the observability reduction be compensated?

These questions guided the work for the main part of the thesis, presented in Part III.

As explained in section 1.3 *"Chronology"*, the area of interest at the beginning of the work on this thesis was fault tolerance mechanisms. The results obtained during this initial stage are presented in Part II.

1.2 Thesis contributions

The main contributions of the present thesis are listed below. A more detailed summary is presented in the conclusions chapter.

- An experimental validation of the feasibility of achieving a low cost safe system by using a mix of already available IP cores.

- A method to detect faults in the cache coherence logic of a multi-core system using a software-based self-test approach.
- A survey of the different solutions that can be adopted to observe the results of functional tests, with focus on in-field test of microprocessor based systems.
- A set of test cases to quantitatively evaluate the benefits and cost of each of the different observability solutions identified.
- A detailed description of the use of a conventional fault simulator to compute the fault coverage achieved by a software-based self-test.
- A set of examples showing that the fault coverage obtained with a test program developed for one observation method may significantly change when reusing it with a different observation method.
- Several observation methods that provide a good coverage of performance faults, a class of faults that are poorly covered by in-field test. Quantitative examples were presented showing that a proper combination of observation methods allow for achieving in-field nearly the same fault coverage achieved when using processor level observation, a method commonly used for production test but usually not available in-field.

In summary, these are results and information that can be fruitfully used both by the test engineer and by the designer.

1.3 Chronology

The collaboration with the thesis advisor started with the work on the master thesis [18], that explored the use of fault injection techniques to evaluate the effectiveness of the fault tolerance mechanisms added to a system. The master thesis produced several publications [19–23] and constitutes a relevant background to the present thesis.

The work has been based in 2-3 month stays of Julio Pérez working with the thesis advisor group at the Politecnico di Torino, interleaved with less intense periods working from Montevideo.

At the beginning, the focus was put on low overhead fault tolerance mechanisms to harden microprocessor systems against single event upsets. First attempts were based on hardware redundancy but remained unpublished. A second approach exploiting time redundancy produced a conference publication [24] that is the basis of the content of chapter 2. All this work was mainly developed during two stays in Turin, one in 2008 and the other in 2010.

Later, the work was reoriented towards the area of in-field test of microprocessor based systems to detect permanent faults. The development of test programs for an in-field functional test scenario was addressed during a first stay in Turin on the second half of 2014. The results of this work were published in a conference paper [25] and are the basis for chapter 4.

Finally, the work on the observation mechanisms for in-field Software Based Self Test [26–29] was started during a new stay in Turin on the first half of 2015 and completed later from Montevideo.

1.4 Thesis organization

The only chapter of Part II is chapter 2 *"Time redundancy fault tolerance"*. This chapter presents some results of the early work on the present thesis exploiting time redundancy in order to tolerate soft errors. A low area overhead solution is presented based on repeating the execution of a task and comparing the results. Proper memory protection is used to confine the fault effects inside only one of the task replicas.

The main part of the thesis is presented in Part III *"Functional test"*.

An introduction to Software-Based Self-Test including a bibliographical revision is presented in chapter 3.

In chapter 4 *"Functional Test of the Cache Coherency Logic in Multi-core Systems"* a method is presented for the test of the cache coherence logic located inside each one of the cores of a multi-core system. A proper test program is run in a coordinated way on each of the cores, enabling the in-field detection of faults. The method was validated and evaluated on a LEON3 multicore system.

In chapter 5 *"Observation Techniques – Survey"* the different solutions that can be adopted in practice to support the observation of fault effects when SBST is adopted for in-field test are presented, with a discussion of the advantages and limitations of each of them.

In chapter 6 *"Observation Techniques – Experimental Results"* we use several test cases to quantitatively evaluate the benefits and cost of each observation solution: one of the test cases targets the branch prediction unit (BPU) in a MIPS-like processor based system, another one targets the cache controller logic in a dual-core LEON3 system, and finally a third test case analyzes the effects on the different modules inside the above mentioned MIPS-like processor.

Finally, some conclusions are presented in chapter 7 *"Conclusions"*.

Esta página ha sido intencionalmente dejada en blanco.

Part II

Fault tolerance – Time redundancy

Part II presents results obtained during the first stage of the work on the thesis, devoted to novel fault tolerance mechanisms. During this stage, the emphasis was on obtaining affordable fault tolerant mechanisms to enable the use of commercial-off-the-shelf processor cores synthesized on FPGAs for the less critical parts of a safety-critical system.

The considered faults were the soft errors provoked by SEUs and the assessment approach was fault injection experiments.

The only chapter in this part essentially consists in the contents of the paper *"Implementing a safe embedded computing system in SRAM-based FPGAs using IP cores: a case study based on the Altera NIOS-II soft processor"* [24]. A low area overhead solution is presented based on repeating the execution of a task and comparing between them the replicated results. Proper memory protection is used to confine the fault effects inside only one of the task replicas. Fault injection experiments were carried out to assess the proposed approach.

Esta página ha sido intencionalmente dejada en blanco.

Chapter 2

Time redundancy fault tolerance

Reconfigurable Field Programmable Gate Arrays (FPGAs) are growing the attention of developers of mission- and safety-critical applications (e.g., aerospace ones), as they allow unprecedented levels of performance, which are making these devices particularly attractive as ASICs replacement, and as they offer the unique feature of in-the-field reconfiguration. However, the sensitivity of reconfigurable FPGAs to ionizing radiation mandates the adoption of fault tolerant mitigation techniques that may impact heavily the FPGA resource usage. In this chapter we consider time redundancy, that allows avoiding the high overhead that more traditional approaches like N-modular redundancy introduce, at an affordable cost in terms of application execution-time overhead. A single processor sequentially executes two instances of the same software; the two instances are segregated in their own memory space through a soft IP core that monitors the processor/memory interface for any violations. Moreover, the IP core checks for any processor functional interruption by means of a watchdog timer. Fault injection results are reported showing the characteristics of the proposed approach.

2.1 Introduction

Today reprogrammable Field Programmable Gate Arrays (FPGAs) are increasingly attracting the attention of developers of safety- and mission-critical applications (e.g., in the aerospace domain) for a number of reasons. First of all, modern FPGA devices offer an unprecedented level of resources (logic, memory, interconnection, arithmetic, and processing resources) that make them highly competitive with ASICs in markets where low production volumes and short time to market are crucial. Secondly, reprogrammable FPGAs offer a competitive advantage with respect to ASICs: when deployed in the field (i.e., in a satellite already in orbit) their configuration can be changed to improve the functionalities they provide (e.g., to change a baseband processing algorithm in a software defined radio system), to correct bugs, to adapt to changing environment conditions, or to implement reconfigurable computing architectures.

Two main technologies are nowadays available for implementing reconfigurable FPGAs: one based on an SRAM configuration memory, where the information defining what functions the FPGA implements is stored on-chip using SRAM memory bits (e.g., Xilinx Virtex devices, and Altera Cyclone devices), and another based on a Flash configuration memory, where the configuration is stored in floating gate cells (e.g., Actel ProASIC devices). When deploying reprogrammable devices in a radioactive environment (such as space), particular care must be posed to the phenomena induced by ionizing radiations, which may impair some functionalities of the circuit the FPGA implements, or even the whole device. In case of SRAM-based FPGAs, ionizing radiation may induce modifications to the configuration information, provoking Single Event Upsets (SEUs) that remain latched until a fresh image of the configuration memory is restored. Conversely, Flash-based FPGAs are immune to configuration memory SEUs. Both SRAM-based and Flash-based FPGAs suffer from radiation-induced SEUs in the memory elements hosted by the reconfigurable fabric (flip-flops, and memory arrays). Moreover, in the case radiation affects the FPGA control logic, effects known as Single Event Functional Interruptions (SEFIs) may be observed, which impair the correct operation of the FPGA until a reset, or a power cycle is performed [30] [31].

To cope with SEUs and SEFIs designers must employ radiation mitigation techniques, which consist in introducing some sort of redundancy in the implemented design or system. The most widely used approach is Triple Modular Redundancy (TMR) [32] [33] that mandates to replicate three times the design (only its memory elements in case of Flash-based FPGAs, or the entire design in case of SRAM-based ones, or even the entire FPGA chip in case of system-level mitigation) and to add majority voters. In case of processor cores implemented on FPGAs, alternative approaches can be exploited to save FPGA resources at the cost of increased computing time. In [34], an approach is presented where the processor core is duplicated, and a custom IP core manages the concurrent execution of the same application on the two cores that work synchronously. Although this approach is effective in reducing the resource overhead with respect to TMR, it still requires processor duplication.

In this work we investigate the possibility to further reduce the area overhead by exploiting time redundancy [32]. According to time redundancy, the same application is executed twice (three times in case we want to achieve error masking), and then an acceptance test is executed. If the two results match, one of the two is forwarded to the user; otherwise, the outputs are discarded, and the computation repeated. An example of application of this concept to processor-based systems for space applications can be found in [35], where the tasks executed by a processor are duplicated, and executed in segregation: each task can access only to its own memory. A custom companion chip takes care of managing any memory access, and guarantees the task segregation.

In this chapter we exploit the concept of time redundancy to develop an architecture inspired by DMT [35] using a soft processor core aiming at being implemented in an SRAM-based FPGA. In order to enforce task segregation, as well as protection against possible SEFIs of the processor core, we resorted to the processor Memory Protection Unit available as soft IP for the selected processor core, and developed two additional IP cores implementing a watchdog timer and a DMA controller.

The main contribution of this chapter lies in the experimental validation of the feasibility of achieving a safe system by using a mix of already available and ad hoc developed (and highly reusable) IP cores, thus minimizing the development time. In our work we exploited the Altera NIOS-II [36] as processor core, and the Altera Memory Protection Unit IP core for memory segregation. Being all the cores already available, and validated, the design effort is limited to the integration with a custom watchdog timer and DMA controller. By exploiting already existing cores a robust system can be obtained, which can be used with a number of different FPGAs supporting the same cores. As a result, a general architecture is obtained which is highly portable and reusable.

To assess the effectiveness of the proposed approach, we also performed a set of fault injection experiments, which show how the SEUs and SEFIs that may affect the processor are effectively mitigated by our architecture.

The rest of the chapter is organized as follows. Section 2.2 presents the adopted architecture. Section 2.3 describes the experiments we performed to assess the soundness of the architecture we developed. Section 2.4 draws some conclusions and outlines future works.

2.2 Adopted architecture

This section describes the architecture we adopted, and details its implementation. The architecture is intended for hardening computing intensive applications executed by a commercial-off-the-shelf processor implemented on SRAM-based FPGAs. The application is supposed to entail a data acquisition phase during which an input buffer is filled with the data that has to be processed, followed by a data processing phase during which an algorithm is applied over the input data and an output buffer is produced, and finally a data presentation phase during which the output data is delivered to the user.

2.2.1 Overview

The architecture focuses on providing protection against transient faults induced by ionizing radiation, the so called Single Event Upsets (SEUs), which may affect the execution of the application by altering the content of processor registers, or by altering the configuration memory of the FPGA. Two protection mechanisms are used.

As far as SEUs affecting the processor memory elements (i.e., register file,

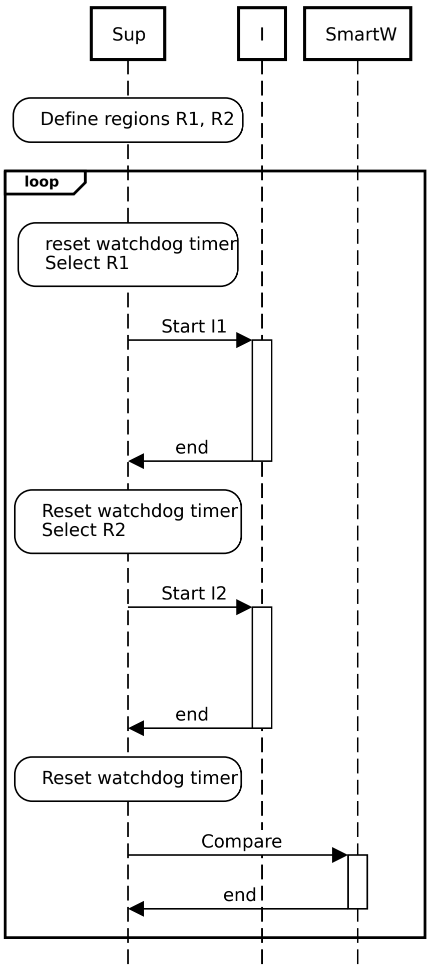


Figure 2.1: Sequence of operations.

special purpose registers, and cache memory) are concerned, task-level duplication is exploited in combination with hardware-assisted consistency check. The application is executed twice, each time writing the results to different memory locations; at the end, the two output buffers are compared. In case of mismatch, the processor undergoes a reset operation, and the whole process is repeated. In case of successful match, the data presentation phase issues to the user the two output buffers: the two buffers are sent to provide a protection mechanism against possible data-transfer errors. In order to guarantee that the two executions of the application instances are performed independently, so that any SEU may affect one and only one of the two executions, a special-purpose hardware module, called *smart watchdog*, is used. The smart watchdog is implemented in the same FPGA used for implementing the processor, and it is placed on the process bus between the processor and the memory hierarchy. It has indeed to snoop for any memory access directed either toward the cache or the main memory. Moreover, the smart watchdog must be able to access to the memory space of the processor to perform the consistency check of the two output buffers.

Let us call I1 and I2 the two instances of the application that are executed sequentially. The processor memory is partitioned in two regions, R1 associated to I1, and R2 associated to I2. Each instance Ix is allowed to read/write only within the boundary of Rx; any access outside Rx indicates the occurrence of an error. The smart watchdog is in charge of monitoring any access the processor performs, and in case Ix is accessing to an address in Ry with x≠y a non-maskable interrupt resulting in the processor reset is activated.

The sequence of operations performed by the processor during the execution of an application is the following:

1. The processor programs the smart watchdog to define R1 and R2, and resets the watchdog timer of the smart watchdog.
2. The processor selects R1 and initiates the execution of I1.
3. Upon completion of I1, the processor resets the watchdog timer.
4. The processor selects R2 and initiates the execution of I2.
5. Upon completion of I2, the processor resets the watchdog timer.
6. The smart watchdog performs the consistency check.
7. The processor resets the watchdog timer and repeats from (2).

Any operation resulting in a wrong memory access outside the region associated to an instance of the application triggers a non-maskable interrupt leading to processor reset. Moreover, any operation leading to a processor hang (i.e., a SEFI) leads to the watchdog expiration, which triggers the processor reset as well. The smart watchdog is in charge of the consistency check: it performs a word-by-word comparison of the output buffers computed by the two application instances. In case of mismatch the processor is reset, otherwise the whole process is repeated. The comparison is implemented through DMA burst transfers that read and compare the two output memory buffers.

As far as SEUs in the FPGA configuration memory are concerned, on-line checking is performed: the configuration memory is constantly read, a checksum is computed and compared with a known-good value. In case of mismatch, the FPGA device is reset, a fresh image of the configuration memory is written to the device, and the whole application is started from scratch.

2.2.2 Implementation

We developed a proof-of-concept implementation of the described architecture on an Altera Cyclone-II device, using the NIOS-II processor core. For the sake of this chapter we focused only on SEUs affecting the processor. The features provided by Altera devices can be straightforwardly exploited to implement the protection mechanism against SEUs in the device configuration memory.

The NIOS-II is a 32 bit, 6-stage pipeline RISC processor. Options include separated instruction and data cache, and a full Memory Management Unit (MMU) or a simpler Memory Protection Unit (MPU). The MPU has separated instruction and data regions. Execution permission can be granted on instruction regions and read or read/write permissions on data regions, both for user and supervisor execution mode. An exception is raised in case of permission violation. Additional exception conditions relevant from a safety point of view are misaligned memory accesses and illegal instruction opcode. The MPU, which is available as IP core, is the building block of our smart watchdog, as described in section 2.2.1 *"Overview"*.

We developed a system encompassing a NIOS-II (version f) core with a 4 Kbytes instruction cache and a 2KBytes data cache. A smart watchdog is attached to the processor encompassing the MPU configured with 6 data regions and 4 instruction regions, an interval timer used as watchdog timer, and a simple DMA controller for output buffer comparison. 512KBytes of main memory are attached to the processor, implemented outside the FPGA device using SRAM chips. Being based on a combination of already existing IP cores, and some custom-made modules, the implementation of the system is highly portable, and it can be mapped on any FPGA device supporting the NIOS-II processor and its MPU. In our implementation we considered Cyclone-II devices, obtaining the resource occupation figures of Table 2.1.

Table 2.1: Resource occupation

Module	**Logic resources** [#]	**Flip-flops** [#]	**Memory bits** [#]
NIOS-II	2 063	1 549	63 104
MPU	963	729	256
Interval timer and DMA controller	350	256	128
TOTAL	3 376	2 534	63 488

With respect to a system including only the NIOS-II processor, the implementation of the proposed architecture leads to the following overheads (due to the addition of the MPU and smart watchdog): 63% of logic resources, 64% of flip-flops, and 0.6% of memory bits. It calls the attention the rather high logic resources usage of the MPU provided by altera, but it is aligned with the resources usage expected by the Nios documentation. As far as the application execution time is concerned, the proposed architecture introduces an overhead of 100%, as two instances of the application have to be executed sequentially. In case the NIOS-II system is implemented using TMR [32], limiting our analysis to the logic resources and flip-flops, we can expect logic resource and flip-flop overheads of at least 200% (not including the resources needed for implementing the majority voters), and at least a 15% in performance overhead due to majority voters added on the processor critical paths. These figures are expected to be even higher in case mitigation techniques for the cache system are considered where the implementation of a protection scheme such as EDAC is expected to impact heavily on the memory bit occupation, logic/flip-flop resources and performance. In the case of the approach presented in [34], we can estimate a logic resource and flip-flop overhead of at least 100%, and a performance overhead of about 30%. Therefore, we can state that the adopted approach, when compared to alternative ones, is effective in reducing the FPGA resource overhead, at a cost of a higher application execution time.

2.3 Experimental results

To assess the robustness of the proposed architecture, we developed a fault injection system, and we performed a set of fault injection experiments. For the sake of this chapter, we focused only on SEUs affecting the processor memory elements. A total of 100 000 SEUs was injected.

We used a software-based fault injection mechanism: we added a SEU injection routine to the code running on the processor, and we used an interrupt request to trigger the injection routine. The mechanism allows injections of SEUs in any software-accessible location within the processor, including general-purpose registers, control registers and program counter. SEUs are randomly injected both in time and space: the injection routine is activated in a randomly-selected clock cycle during application execution, and a SEU is injected in one randomly selected bit of a randomly selected register. The fault injection process encompasses the following operations:

1. The FPGA is configured, a fresh image of the processor memory is downloaded, and the processor is reset. Except for the FPGA configuration, this initialization operation is performed for each fault in order to guarantee that experiments are independent from each other.

2. Through the debug interface the memory location storing the injection time, and the injection location (register and bitmask to use) are modified according to the fault to be injected.

3. A timer initially set to the injection time is started, and the application execution is started according to the mechanism described in section 2.2.2 *“Implementation”*.

4. Upon expiration of the injection-time timer, the application is stopped, and the SEU injection routine is activated; as a result, the desired fault is inoculated in the system.

5. The execution of the application is resumed until its completion.

At the end of the application execution, the two output buffers are analyzed, and SEUs are classified as follows:

- *No effect*: The execution completed successfully, the two output buffers contain the same values and match the results produced by a fault-free execution. Moreover, the MPU did not trigger the non-maskable interrupt, and the watchdog timer did not expire.
- *Data detection*: The execution completed successfully, but the output buffers produced by the two application instances have different values, while the smart watchdog signaled the mismatch.
- *Exception*: The smart watchdog triggered the non-maskable interrupt, signaling that an attempt to access a forbidden memory partition is detected.
- *Timeout*: The smart watchdog timer exhausted before the end of application execution.
- *Trap*: The injected fault triggered one of the processor traps, indicating that an erroneous situation is detected (e.g., the processor is executing a misaligned memory access).
- *Wrong answer*: The execution completed successfully, the two output buffers are equal but they do not match those produced by a fault-free execution. Moreover, the MPU did not trigger the non-maskable interrupt, and the watchdog timer did not expire. This condition corresponds to the case in which the fault produced a misbehavior, but escaped all the error detection mechanisms our architecture offers.

In our experiments we considered a data-processing benchmark composed of a 16-tap finite impulse response filter processing 512 samples of a 500Hz tone sampled at 8KHz. The filter is implemented in the direct form, and its coefficients are chosen to obtain a 1KHz low pass filter. Then, a further processing based on computing the square root of the obtained values is executed. Benchmark execution, according to the sequence of operations described in section 2.2.2 *"Implementation"* lasts for about 8 million clock cycles.

During preliminary injection experiments to tune the system, some faults were identified that put the processor in a halt state, leading to a SEFI condition detected as timeout. By analyzing these faults, we found that the halt condition is the result of a fault leading to a jump into an unused code area where the custom instruction opcode is found (resulting from random initialization of the memory). The NIOS-II processor has a reserved "custom instruction" opcode to enable instruction-set extensions through the addition of custom hardware. The opcode for the custom instruction is predefined and it is not detected as illegal, even if the processor is not equipped with the custom hardware to execute this instruction. To avoid this situation we initialized the whole unused memory with an illegal opcode. In this way, any fault causing a jump to a word in the unused memory triggers an exception, and thus can be detected.

Once the system has been set up and tuned, we run a preliminary set of fault injection experiments during which we injected 100 000 SEUs in the processor program counter. These experiments were useful for getting an initial indication of the soundness of the approach by considering very critical faults, which may dramatically harm the health of the system. The classification of fault effects is presented in Table 2.2.

Table 2.2: Clasification of fault effects

No effect	6 178
Data detection	8 498
Exception	63 110
Timeout	0
Trap	22 161
Wrong answer	53
Total	100 000

From these results we can see that 53 of the injected faults (0.053 %) escaped the detection mechanisms the architecture embeds. Moreover, by looking in more details to the results, we observed that the following traps are executed:

- 9 029 illegal opcode traps, indicating the effectiveness of properly initializing the unused memory, which is likely to allow avoiding the timeout condition;
- 4 448 misaligned data address trap;
- 8 683 misaligned destination address trap;
- 1 supervisor only instruction trap.

The results suggest that the processor already embeds very powerful mechanisms for detecting misbehaviors induced by SEUs.

We also observed that SEU effects are strongly related to the position of the fault in the Program Counter. All the faults affecting bits 0 or 1 on the Program Counter (the least significant bits) were detected by misaligned memory access traps. On the other side, all the faults affecting the highest order bits of the program counter provoked accesses out of the memory regions configured on the MPU, and consequently were detected as region violation exceptions. For the intermediate range of bits, the fault effects gradually change from a majority of no effect and data detection to a majority of illegal instruction and region violation exceptions as the affected bit varies from lower to higher order bits.

2.4 Chapter conclusions

As reprogrammable FPGAs become the devices of choice for developers of safety- or mission-critical applications operating in radioactive environments, suitable mitigation techniques are needed against soft errors originated in the FPGA devices. In this chapter, we describe an architecture that exploits task-level redundancy in combination with already available IP cores for implementing a processor-based system robust with respect to SEUs. Preliminary results focusing on the processor program counter outline the effectiveness of the architecture, and the overhead analysis shows that the proposed architecture is effective in reducing the resource occupation with respect to N-modular redundancy, at a moderate cost in terms of application execution time. The reported results allow a deeper understanding of the fault behavior and of the effectiveness of the different fault detection mechanisms.

Part III

Functional test

This part introduces the central aspects of the thesis and is organized as follows.

An introduction to Software-Based Self-Test including a bibliographical revision is presented in chapter 3.

In chapter 4 *“Functional Test of the Cache Coherency Logic in Multi-core Systems”* a method is presented for the test of the cache coherence logic located inside each one of the cores of a multi-core system. A proper test program is run in a coordinated way on each of the cores, enabling the in-field detection of faults. The method was validated and evaluated on a LEON3 multicore system.

In chapter 5 *“Observation Techniques – Survey”* the different solutions that can be adopted in practice to support the observation of fault effects when SBST is adopted for in-field test are presented, with a discussion of the advantages and limitations of each of them.

In chapter 6 *“Observation Techniques – Experimental Results”* we use several test cases to quantitatively evaluate the benefits and cost of each observation solution: one of the test cases targets the branch prediction unit (BPU) in a MIPS-like processor based system, another one targets the cache controller logic in a dual-core LEON3 system, and finally a third test case analyzes the effects on the different modules inside the above mentioned MIPS-like processor.

Even when the concepts and techniques presented here are applicable to other fault models and scenarios, most of the examples and experimental cases introduced in the following chapters aim to detect permanent stuck-at faults in an in-field scenario.

Esta página ha sido intencionalmente dejada en blanco.

Chapter 3

Background on Software-Based Self-Test

When electronic systems are used in safety critical applications (e.g., in the space, avionic, automotive or biomedical areas), we need to guarantee that the probability of failures due to faults of any kind (unreliability) is lower than a specified threshold. One of the possible causes of failures are defects affecting the hardware components. Different techniques can be used to reduce the chance that hardware defects can occur (e.g., acting on the adopted semiconductor technology) or to minimize the probability that they produce critical failures (e.g., by introducing redundancy).

Unfortunately, most of these techniques have a severe impact on the cost of the resulting product. In some cases (especially when advanced semiconductor technologies are used, e.g., to achieve enough performance) the probability of failures is anyway too large.

Hence, a solution which is commonly used in several scenarios lies on periodically performing a test able to detect the occurrence of any fault before it produces a failure (*in-field test*). This kind of test can resort either to Design for Testability (DfT) or to functional approaches.

The latter solution is normally based on forcing the CPU inside the Device Under Test (DUT) to execute a properly written test program, which is able to activate possible faults and to make their effects visible in some observable locations (e.g., the data memory). This approach (also called *Software-based Self-test*, or SBST [4]) has the advantage of testing the DUT exactly in the same conditions of the final application, and is more suitable for concurrent in-field test, since it is less intrusive than DfT. On the other side, SBST solutions may require a large effort to develop a suitable test program.

The SBST approach is currently experiencing a growing success, mainly because it offers the possibility (besides the other advantages) to the semiconductor company manufacturing the device (and knowing its internal structure) to develop the test code, grade it in terms of achieved Fault Coverage, and pass it to the system company, which eventually integrates it in the application code. The system company is also in charge of developing the code in charge of launching the test and retrieving the results it produced, managing the situations where a fault is detected.

Since the test code is often activated in small chunks, whose execution can fit in the idle times of the application, it is convenient to organize it in a set of procedures, composing a *Self-Test Library* (STL). STLs are currently offered by several semiconductor and IP companies [7] [8] [9] [10] [11] [12].

When developing the code of a STL, special techniques must be followed to activate the target faults and to make them visible. The latter point is particularly important, especially because during in-field test the observability of the DUT behavior is necessarily limited. Hence, several solutions can be adopted, possibly involving the support of existing or ad hoc hardware [28].

When considering permanent faults that may affect an electronic device (such as a microprocessor, or a System on Chip), a special class is represented by *Performance Faults*, i.e., those faults which do not affect the results of the computation, but simply the timing behavior of the DUT. Examples of these faults can be found in a Branch Prediction Unit (BPU). If the BPU is faulty, it may always produce a wrong branch prediction. The final result of the program execution will be correct, but the time required for the execution will be larger. Performance Faults can be found in other parts of a processor, such as the cache and memory controller. Clearly, the relevance of Performance Faults from a practical point of view may change depending on the application. Since real-time constraints are often important, in many cases they cannot be neglected and need to be detected.

Some previous works already dealt with Performance Faults. In particular, in [16] the authors describe a method to detect them resorting to Performance Counters, i.e., those hardware structures which are often included in a processor to count the occurrence of internal events (e.g., cache misses, or wrong branch predictions), mainly to support the manufacturer in assessing the correct behavior of the design. In [37] a method to estimate the impact of performance faults on different performance-related modules in a CPU is proposed. Performance counters and their usage for testing purposes are revisited in section 5.5.

The term Software-Based Self-Test (SBST) was first proposed by Chen and Dey in [38], but the approach itself has been proposed few years before under the name “Native Mode Functional Test” in [39] and [5]. SBST broadly identifies all test methodologies based on forcing a microprocessor/microcontroller to execute a program and checking the results to detect the presence of possible defects affecting the hardware. Indeed, the pioneering idea of testing a microprocessor with a program dates back to 1980. In [40], Thatte and Abraham devised fault models and procedures for building test programs able to detect permanent defects in different functional units of a simple processor. A wide adoption of their methodology was hindered by the difficulties in automating the generation of such test programs, especially when targeting complex processors.

In general, the usage of SBST requires:

1. Generating a suitable test program. This is typically a hard job, which is still mainly performed by hand. Moreover, the complexity and effectiveness of this task depends on the adopted metric, which in turn depends on the available information: in some cases, both RTL and gate-level models of the target system are available, while in others functional information is available, only. For the purpose of this work, we assume that the gate-level netlist is available unless it is explicitly stated, and it is possible to compute the fault coverage achieved by the generated test program with respect to the most common structural fault models (e.g., stuck-at). In chapter 4 *"Functional Test of the Cache Coherency Logic in Multi-core Systems"* the development of a test program targeting some specific faults is addressed.
2. Creating an environment to support its execution. Once the test program is available, it must be stored in some memory accessible by the processor, the processor must be triggered to execute it at the due time, and the results produced by the processor during its execution must be observed. In chapters 5 and 6 we specifically focus on the last issue.

Nowadays, the complexity of processors has significantly increased; the micro-architectural details play a fundamental role, and devices cannot be accurately modeled using information about the Instruction Set Architecture (ISA) alone. However, SBST is becoming more and more important: it commonly supplements other kinds of tests, as functional programs may detect unmodeled defects that escape to traditional structural tests (the so-called "collateral coverage" [41]). By definition, the functional approach tests the system in its operational mode, without activating a test mode and without reconfiguring the system; hence, it is guaranteed not to cause overtesting (or overkilling). Moreover, several producers exploit functional stimuli to validate their design or to run post-silicon verification.

In some cases, test programs are generated pseudo-randomly [42], possibly using simulation feedback, and may even use some hardware support to make more efficient the test phase [43].

Amongst the several recent works focusing on SBST, some aim at developing algorithms to generate effective test programs for common modules starting from the knowledge of its ISA alone (e.g., for an OpenSPARC T1 processor [44] or a MIPS-like ISA [14]), eventually combined with RTL description (e.g., [45] for two different implementations of the MIPS ISA). Others focus on the possible automation of the test program generation procedure (e.g., [46] using a LEON2 processor, [17] using miniMIPS). The possible usage of SBST for diagnostic purposes has also been explored (e.g., [47] using an 8-bit accumulator-based microprocessor, and [48] for transition-delay faults in an i8051-compatible microcontroller). Finally, a number of works study how to apply SBST for in-field test (e.g., [49] for a MIPS architecture processor).

In this last domain, regulations and standards mandate the adoption of effective solutions to early detect permanent faults, and SBST has the big advantage that it does not require access to any systematic DfT solution (such as scan or BIST), whose usage details are often considered as proprietary by the manufacturer. SBST can be used not only to test the CPU, but also the other components in a microcontroller or SoC: for example, several works addressed its adoption for the test of peripherals [50], memories [51] (possibly implementing transparent test [52]) and cache memories [53] [54]. SBST usage can also be extended to the test of multi-core systems [55]. In some cases, the usage of existing hardware resources introduced for non-test-related purposes (e.g., for debug, design validation, performance assessment) allows significantly reducing the size and duration of SBST test programs [56].

Moreover, SBST can more easily match the constraints of the environment where the processor is employed. When adopted for in-field test, SBST typically means activating a test program either at the system power-on, or during the application idle times. In the latter case, additional constraints about the duration of the test exist, due to the limited duration of the available time slots. Unfortunately, the constraints posed by the application environment may severely impair the effectiveness of the method when applied to test a processor. When functional test is used for end-of-manufacturing test, processor inputs and outputs can often be fully controlled and observed by an ATE. Nevertheless, during in-field test some parts of the processor (e.g., the test and debug structures) might not be accessible by the test procedure, thus resulting in untestable faults [6], i.e., faults for which no input stimuli exists, able to detect them. In other words, some parts of a processor which are not used anymore during the operational phase may contain faults, which cannot be tested in this phase, but are also guaranteed not to affect the system behavior. Besides, not all the processor inputs may be freely controllable in the in-field scenario: for example, activating the reset signal is hardly possible, and thus the test of the reset logic is prevented. More in general, possible Control/Status input signals coming from other devices may be hard to control [5]. Finally, observability during SBST in-field test could be limited, since only the produced results (e.g., in memory) can be observed. The set of faults which cannot be tested in the in-field environment due to these additional constraints are known as functionally untestable faults [6]. As previously mentioned, it is important to be able to identify untestable faults, since they limit the achievable fault coverage.

In the following chapters several aspects of Software-Based Self-Test are presented. The chapter 4 presents an example of the development of a functional test in an in-field scenario. A method to test the cache coherency logic embedded in each processor's cache in a multiprocessor system is presented together with an evaluation of its cost in terms of execution time.

Then we focus on the observability issue: firstly, in chapter 5 *"Observation Techniques – Survey"* we describe in detail different observation solutions that can be adopted for SBST in-field test, comparing them with the ideal case in which all the outputs of the module under test (or of the processor) are continuously observable during the test. Secondly, in chapter 6 *"Observation Techniques – Experimental Results"* we use several test cases to quantitatively evaluate the loss in fault coverage that stems from the adoption of the different solutions.

Esta página ha sido intencionalmente dejada en blanco.

Chapter 4

Functional Test of the Cache Coherency Logic in Multi-core Systems

Multi-core systems are becoming particularly common, due to the high performance they can deliver. Their performance strongly depends on the availability of effective cache controllers, able to guarantee (among others) the coherence of the caches of the different cores.

Here, a method is proposed for the test of the cache coherence logic existing within each core in a multi-core system, resorting to a functional approach; this means that the method is based on the generation of a suitable test program, to be run in a coordinated manner on the cores composing the system. The method is able to detect hardware defects affecting this logic. The method was validated and evaluated on a LEON3 multicore system. Most of the content of the present chapter was already published in a conference paper [25].

4.1 Introduction

Multi-core systems are increasingly popular in the applications where high performance is required, due to the interesting mix of performance, flexibility and power they offer. However, the complexity of the devices implementing such multi-core systems, combined with the increased susceptibility to faults of new technologies, asks for new techniques able to effectively detect possible faults affecting their hardware structure, both at the end of the manufacturing process, and during the operational life (in-field test).

A common solution lies on resorting to Design for Testability (DfT) techniques, such as scan test or Logic BIST. However, these techniques may sometimes be inadequate: firstly, these solutions may often not be exploited during the operational life, for example because they require an external tester (not available for in-field testing); secondly, because IP producers tend not to disclose details about the DfT architectures, avoiding to impair IP protection; thirdly, because sometimes DfT is inadequate to achieve sufficient defect coverage, that can only be obtained by running the test in the same operating conditions (e.g., in terms of speed) and configuration of the application. For all these reasons, a functional test approach based on developing suitable test programs to be executed by each core and on observing the results produced is a suitable solution. As mentioned before, this approach is also known as Software-Based Self-Test (SBST) [4].

Caches are one of the most critical components within multi-core systems, since their behavior can seriously affect the performance of the whole system. Previous papers [53] [57] [58] already described how their data and tag parts can be effectively tested resorting to a SBST approach. In some cases, their test can be made easier by exploiting the special instructions provided by some Instruction Set Architectures to directly access their content [59].

Additionally, multi-core systems require proper coherence protocols, able to guarantee that the content of the caches of the different cores is always up-to-date, so that each time a processor accesses a piece of data, it always accesses a correct and coherent value. Validation of cache-coherent multiprocessors is a challenging task, often performed through extensive simulation of randomly generated sequences of operations [60] [61]. On the other side, it is also crucial to check whether any hardware defect affects the cache coherence logic. In [62] the focus was on the test of the coherence logic of a cache controller implementing the Modified-Exclusive-Shared-Invalid (MESI) protocol. In that paper, it was only considered the logic corresponding to the Finite State Machine implementing the protocol, neglecting the rest of the involved control circuitry.

The purpose in the present chapter is to propose a method to generate a proper test program to be run on a multi-core system in order to check whether the circuitry implementing the cache coherence protocol is affected by any hardware fault. The test program is derived from the functional specifications of the circuitry under evaluation only, and can therefore be reused on any circuit implementing the same coherence protocol. Interestingly, since the proposed technique is based on a test program, it is well suited to be adopted by system companies for both Incoming Inspection [63], and in-field test (since it is possible to activate the test at any time during the operational phase).

In order to practically validate our approach and to better quantify its cost in terms of memory occupation and execution time, some experimental results were gathered using a multi-core system based on the LEON3 processor [64].

4.2 Background

Nowadays, multi-core systems usually include multi-level caches. Each cache has a corresponding cache controller, implementing not only the functions required to properly operate the cache by itself, but also to guarantee the coherence of the shared data allocated at a given time on the different processors' caches.

In particular, the Cache Coherence Logic (CCL) mainly aims at avoiding the case in which two copies of the same memory block allocated in two different caches do not contain the same values. To avoid this problem, several mechanisms exist. One of the most popular, which is considered here, is based on spying the addresses flowing on the bus (snooping), so that a block in a cache is invalidated if the value of the same block has been changed in another processor cache. Hence, a key role in the cache coherence logic is played by the Validity Bit (VB) associated to each cache line. The VB is substituted by several bits when the adopted cache coherence protocol is more complex, like in the case of the MESI protocol [65].

In this chapter, we consider a Cache Coherence Logic implementing a simpler protocol, such as the one adopted for the data cache of the LEON3 processor core [64]. In such a case a Validity Bit is associated to every cache line; in addition, the cache implements the write-through, no allocate mechanism. If the processor is used in a multi-core configuration, the cache coherence logic continuously snoops the bus transfers: if another processor executes a write operation on a block which is also stored in the local cache memory, the block is invalidated (i.e., VB is forced to 0) thus forcing every further access to the block to access the memory. VB is forced to 1 each time a new block is uploaded into the cache memory.

Based on the above discussion, the CCL is mainly composed of the following elements:

- the VBs (one for each cache line);
- a set of comparators, whose inputs are the external address bus and the tag fields associated to the cache set corresponding to the address currently on the bus (one for each of the possible ways a given block can be stored in a set associative cache);
- some control logic, able for example to interact with the bus and understand when to sample the address value during a memory write operation.

In the next section we will propose an algorithm able to test these three pieces of circuitry by executing a proper test program and checking the system behavior.

4.3 Proposed approach

It is described here the algorithm proposed to test the cache coherence logic in a multi-core system; for the sake of simplicity the usage of the algorithm in a dual-core system where each core includes a direct mapped cache is initially discussed. These assumptions will be removed in section 4.3.4 *"Optimizing the test in multiple-core systems"*. It is also assumed that a previous test has been run, able to test the cache itself including the cache controller circuitry not corresponding to the coherence logic.

The algorithm targets stuck-at faults. The test of the targeted logic requires first exciting each fault, and then observing the fault effects. We will deal with the two issues separately. It is important to note that the basic function of the CCL is to invalidate a given cache line when another processor executes a write operation on the memory block it stores, i.e., to properly modify the value of the corresponding VB. Hence, observing the effect of any fault in the CCL once it has been excited means observing the value of the corresponding VB.

In order to perform this set of operations, in the following the different memory operations involved in the CCL testing are described. They require the use of two processor cores: P0 and P1. P0 is the target processor core whose CCL we want to test, whereas P1 is a support processor intended to execute operations that invalidate the data in the P0 cache module.

4.3.1 Excitation phase

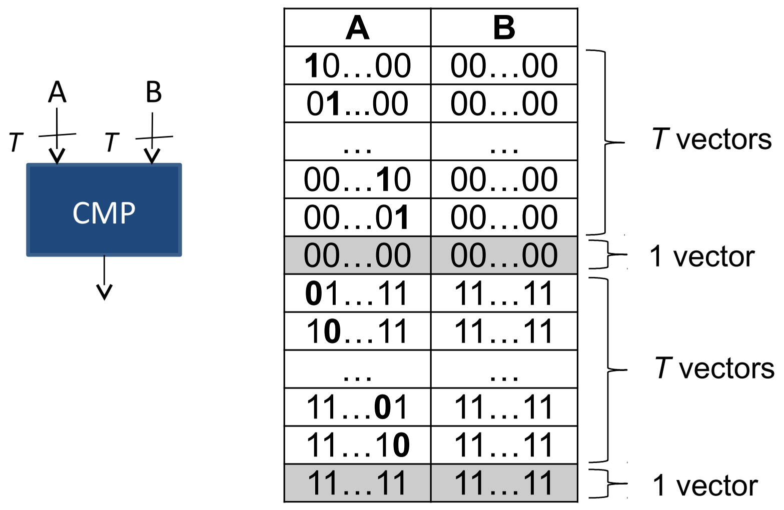
We first need to check whether any stuck-at fault exists, affecting the VBs.

In the following, the required operations developed for this purpose are detailed. Every step details the processor required to execute the listed operations and the expected behavior in the targeted cache:

0. P0 - cache flush; all validity bits are initialized to 0;
1. P0 - upload each cache line with a known block (thus turning all VBs to 1); for every line a read operation is performed and a cache miss is expected;
2. P0 - access the block which was uploaded in each line in the previous step, checking that a hit is triggered; if not, the corresponding VB is affected by a stuck-at-0;
3. P1 - invalidate the P0 cache (thus turning all VBs in P0 to 0);
4. P0 - access the block which was uploaded in each line. In the absence of faults, all VBs turn back to 1, and a cache miss is expected; if not, the corresponding VB is affected by a stuck-at-1.

Each of the above steps (except step 0) consists of nS read or write operations, being nS the number of sets in the cache (equal to the total number of cache lines in a direct mapped cache), each operation accessing to a memory location which is supposed to be stored in a different cache line. Details about the rules to compute these addresses can be found in [53]. Hence, the above steps require $4.nS$ instructions, plus the cache invalidation instruction (*flush* in the case of the LEON3 assembly code).

Secondly, we need to check whether the CCL is affected by any fault. In a direct mapped cache, the CCL is basically composed of a comparator; each time a processor core accesses the memory, this comparator compares the tag portion of



A	B
10…00	00…00
0**1**…00	00…00
…	…
00…**1**0	00…00
00…0**1**	00…00
00…00	00…00
01…11	11…11
1**0**…11	11…11
…	…
11…**0**1	11…11
11…1**0**	11…11
11…11	11…11

Figure 4.1: Comparator schema and test patterns.

the address on the bus with the content of the tag field of the corresponding cache line. For testing the comparator we can exploit the algorithm proposed in [66]. Calling T the number of bits in the tag field, such an algorithm specifies a set of $2.T+2$ input vectors that should be applied to the $2.T$ comparator inputs (as shown in Figure 4.1), guaranteeing that they allow achieving full stuck-at fault coverage, independently of the specific comparator implementation.

Applying each of these test patterns to a comparator in the CCL requires the following steps:

1. P0 - upload in a suitable cache line a memory block, so that the value of the tag field matches the required value (input B); this can be achieved by executing a read access to a suitable location in memory;
2. P1 - execute a write operation on the block uploaded at the previous step; this implies that the required value is written to the bus, and thus applied to the A input of the comparator.

The first step can be ommited when the value of input B is the same as in the previous test pattern, because the tag value remains unchanged in the allocated cache line. Depending on the test vector, the comparator is expected to produce a match or mismatch; correspondingly, the related VB in P0's cache is forced to 0 or left at 1. The full sequence is shown below, detailing for each item the expected activity on the shared bus:

- P0 reads a value from an address with the tag field equal to all zeroes. This uploads a memory block to a cache line and sets the value of comparator's B input for the first $T+1$ test patterns (1 read transfer).

- P1 produces T write transfers to the proper addresses to provide the A values for the first T test patterns. The comparator is expected not to match, maintaining the validity bit set (T write transfers).
- P0 reads again the initial value, a cache hit is expected (no activity).
- P1 writes to the memory block, providing the A value for test pattern number *T+1*. A comparator match is expected, thus making P0 to invalidate the corresponding cache line and force the VB to 0 (1 write transfer).
- P0 reads again, but now a miss is expected (1 read transfer).
- The whole sequence is repeated for an all ones tag value.

The algorithm can be easily extended to a core including an nL ways set associative cache. In this case the CCL includes nL comparators, and the algorithm should be repeated to test each of them.

4.3.2 Observation phase

The above algorithms allow forcing a known VB to 0 or 1. In order to observe whether the target VB holds the expected value or not, the test program should execute an access to the block stored in the corresponding cache line. If this triggers a hit, it means that the VB holds the value 1, otherwise (miss) the VB holds the value 0. Most of the faults affecting the CCL can be labeled as *performance faults*, i.e., they do not affect the correctness of the result produced by the test program, but rather its performance [67].

In order to observe whether a given memory access triggers a hit or miss we can adopt different techniques, based on the hardware mechanisms available:

- *performance counters* existing in the processor, devoted to count the number of hit and miss situations triggered by a given program [16] [68]
- an *internal timer*, devoted to measure the test program execution time
- the *debug infrastructure* usually provided by the processor, able to trace and communicate to the outside the bus activity for a given period [69]
- an *ad hoc module* added to the system and in charge of monitoring the bus activity [70]

4.3.3 Analytical performance analysis

The main component of the test time required by the algorithm is related to memory accesses. Their number can be estimated as shown below.

Some of the transfers needed for the execution of the test are cache hits and consequently internal to a core, while others are cache misses and will compete with the other processors to access the bus. In both cases the amount of necessary data transfers depends on the number of cores *ncpu*, the number of tag bits T (which affects the size of the required comparator), and the number of sets nS in the cache, assuming it is direct mapped. The activity in the external shared bus corresponds to all the write accesses and the missed read accesses. The number of transfers required by the complete test are:

$$Nshared = [nS * (2rd + 1wr)] + [T * 2wr + 4rd + 2wr]$$

The terms into the first bracket correspond to the test of the validity bits while the second bracket corresponds to the test of the comparator logic.

The internal transfers (read accesses resulting in a cache hit) on each core are:

$$Ninternal = (nS + 2) * rd$$

4.3.4 Optimizing the test in multiple-core systems

In the previous sub-sections we described how to test the possible stuck-at faults affecting the CCL of a target processor core, using a second core as a support.

In the case of an *ncpu*-core system some parallelism can be exploited. A first approach consists in using each processor to support the test of the following one. Processor P0 plays the role of support processor for P1, P1 for P2, and so on. This approach was the one used to obtain the experimental results presented in the next section. The total number of transfers in the shared bus is multiplied by *ncpu*.

$$Nshared = ncpu * ([nS * (2rd + 1wr)] + [T * 2wr + 4rd + 2wr])$$

The total duration of the algorithm will depend on the performance of the bus. In an ideal scenario where there is no bus contention, the test duration will remain constant, equal to *T1* the duration of the algorithm for testing a single core. On the other side, if the memory bus access is saturated the execution time will be dominated by the memory accesses and will increase linearly with the number of processors *ncpu*T1*.

A second approach that saves some memory transfers uses only two support processors. For example, processor P0 plays the role of support processor for P1 and all the other odd numbered processors, and P1 for P0 and all the even numbered processors. This approach limits the number of processors doing write operations to 2 instead of *ncpu*, and consequently reduces the bus traffic. The resulting total number of transfers in the shared bus is:

$$Nshared = [ncpu * nS * 2rd + 2wr] + [ncpu * 4rd + 2 * T * 2wr + 2 * 2wr]$$

4.4 Experimental results

The approach effectiveness has been experimentally evaluated in a multi-core system based on the freely available LEON3 processor by Aeroflex Gaisler [64].

A multi-core system was implemented including a minimum set of memory cores, plus a configurable number of LEON3 processors. Every processor core is instantiated with separate data and instruction caches. The configuration for the data caches used in our test was 1 way (direct mapped), 1Kbyte/way, 16 bytes/line. As mentioned, the data cache implements the write-through policy, with no-allocate on a write miss.

The test program on each processor requires the execution of 317 assembly language instructions for the VB test and 412 instructions for the Comparator test part, plus some loop instructions for synchronization. For every processor core, the compiled test program requires about 1KB of code memory.

The execution times for both parts of the test (VB and Comparator) are reported in Figure 4.2 for systems with 2 to 8 processors. VB test corresponds to the solid line, and Comparator test to the dashed one. All values are expressed in number of clock cycles. These values show that the effects of bus and memory contention do impact more significantly on the VB test, where the execution time grows more quickly with the number of cores. In the Comparator test the execution time grows more slowly, due to the higher amount of parallelization that our algorithm achieves.

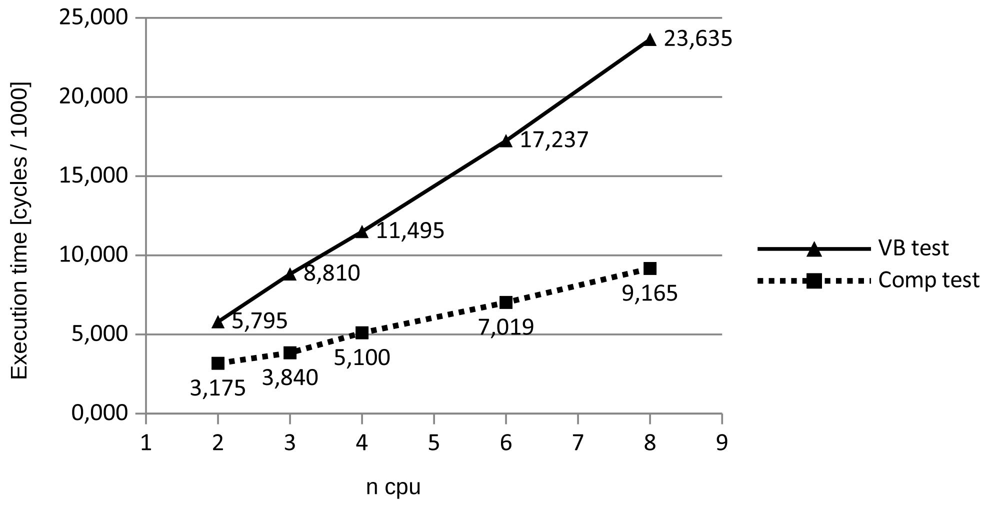


Figure 4.2: Test programs execution time (clock cycles) vs. number of cores.

In order to validate the correctness of the proposed algorithm we analyzed the LEON3 data cache controller RTL source code to identify the parts of it which implement the snoop mechanism. The whole system was then simulated at the RT level using the Mentor Graphics ModelSim tool to check that the algorithm behavior is the expected one.

On the other hand, it was not possible to identify the cache controller logic as a piece of circuit separated from the rest of the cache controller at the gate level. For this reason it was not possible to obtain a list of the faults associated to the cache coherency logic and as a consequence no fault simulation experiments were performed.

The prototype implementation that was evaluated does not include a mechanism to observe if a given memory access produces a cache hit or a cache miss. A final implementation must include such a mechanism and depending on the chosen mechanism it may introduce some area or configuration time overheads.

4.5 Chapter conclusions

In this chapter we propose a method to detect possible faults affecting the hardware that implements the cache coherence logic integrated in each cache controller of a multi-core system.

The method is based on a functional approach, i.e., on the execution of a carefully written test program executed by different cores in a coordinated manner. The method achieves by construction a full fault coverage of the static faults in the main components of the addressed logic, namely the validity bits and address tag comparator; it is suitable to be used both during end-of-manufacturing test and for in-field test (e.g., when safety-critical systems are considered).

We experimentally evaluated the proposed approach on a system integrating a variable number of LEON3 cores, showing its cost in terms of execution time, which grows linearly (and slowly) with the number of cores.

Esta página ha sido intencionalmente dejada en blanco.

Chapter 5

Observation Techniques – Survey

The main purpose of this chapter is to survey the different solutions that can be adopted in practice to support the observation of fault effects when SBST is adopted for in-field test, discussing the advantages and limitations of each of them. In the next chapter we use several test cases to quantitatively evaluate the benefits and cost of each observation solution: one targets the branch prediction unit (BPU) in a MIPS-like processor based system, another one targets the cache controller logic in a dual-core LEON3 system, and finally a third test case analyze the effects on the different modules inside the above mentioned MIPS-like processor.

A comparison of the fault coverage obtained using two different observation methods is presented in [71]. However, to the best of our knowledge, the present work and its associated papers [26] [28] are the first report of extensive experimental results to compare the fault coverage that can be achieved with the different observation methods, thus allowing the reader to have a better understanding of the advantages and disadvantages provided by the different solutions. These two chapters also outline some techniques to compute fault coverage figures related to the usage of an SBST approach with different observation mechanisms.

In the following, the main solutions that can be adopted to observe the effects of possible faults during in-field SBST testing of a processor-based system are described, namely: module-level, processor-level, system bus, memory content, performance counters, and debug interface. The above list also includes a few solutions that only represent ideal solutions taken as reference, although they can hardly be adopted during in-field SBST. The scope of the present work is limited to *bare metal systems*, i.e., solutions that would require the presence of an Operating System (e.g., based on monitoring its performance, or analyzing the event logs) were not considered.

It is assumed that the targeted faults are those inside a given module within the processor. For every solution, the adopted mechanism as well as the main advantages and disadvantages are detailed, and a preliminary analysis about the forecasted coverage is reported. Figure 5.1 summarizes some of the considered observation solutions, referring to the architecture of a simple processor-based system.

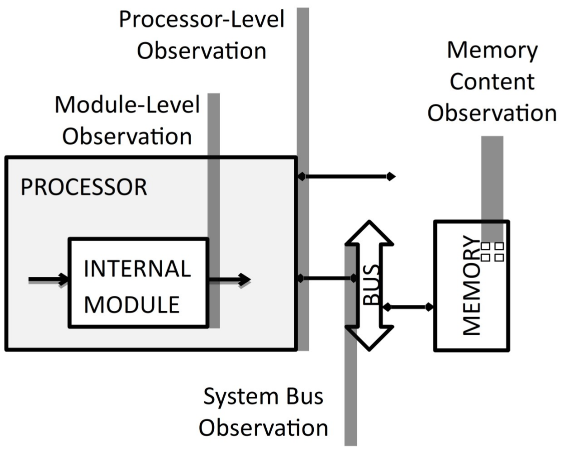


Figure 5.1: Generic system under test: the observation points adopted by the first four of the techniques described in the text are highlighted.

5.1 Module-Level Observation

When a generic module inside the processor is considered for the test, the ideal level of observability is the boundary of such a module, i.e., it is assumed that all the output ports are available for observation.

The test program is supposed to be able to properly stimulate the input ports of the considered module, to excite the faults and then to propagate them towards the module output ports, which are test observation points.

This observation solution can be adopted during simulation and fault simulation processes. However, when working on real chips, for a number of reasons it is hardly feasible neither in-field nor, in most cases, at end-of-manufacturing. These reasons are, firstly, that the module output ports usually do not coincide with the circuit pinout, and even if they do, it is normally not affordable to continuously observe the circuit behavior without resorting to additional hardware; secondly, when an instrument is attached to the observation points, the observed signals can only be read in test mode through a dedicated tester.

Therefore, this solution is introduced here only as a reference, because it establishes an upper bound to the fault coverage figure obtainable through SBST test approaches.

5.2 Processor-Level Observation

This solution assumes that fault effects can be observed at the processor level, i.e., that all the processor outputs can be continuously monitored. While module-level observation is very specific and may be not feasible in practice, observation at processor-level represents one of the scenarios that are sometimes adopted for end-of-manufacturing test. Considering an internal module which has to be tested, the test program must not only propagate the fault effects up to the module output ports, but must be able to propagate them also to the processor output ports.

According to these considerations, the observability we can get with this solution is generally lower than the one obtained at module-level. In most of the cases, propagating the faulty behavior requires an additional effort in order to reach the processor outputs. In the case of a functional testing approach based on test programs, this additional effort may imply the addition in the code of specific instructions able to propagate the fault effects to the processor outputs. As an example, faults within an arithmetic unit can be easily activated by executing suitable arithmetic instructions (thus propagating their effects on the module outputs), and can then be made observable on the processor outputs via store instructions that propagate the result of the arithmetic operation up to the processor output ports.

Faults may also exist that, even with the addition of instructions, cannot be observed on the processor outputs. This situation may happen when the processor design includes some redundant circuitry, for example left from previous releases or included for future extensions of the design. Clearly, the related faults can be classified as untestable. However, the identification of untestable faults may often represent a relevant problem.

Due to the need of constant monitoring of all the processor outputs, this observation solution requires the use of an ATE and thus, cannot be adopted by in-field SBST.

5.3 System Bus Observation

This solution mandates that the control, data and address signals of the system bus are continuously monitored. When comparing this solution with the previous one, all the processor outputs not related with the system bus are excluded from observation.

End-of-manufacturing scenarios may offer a high level of observability when the constant monitoring of the output ports of the processor is possible. Such a powerful scenario is not representative of an observation mechanism for in-field testing. However, more and more processors (especially for embedded systems) are equipped with specific components in charge of monitoring the interconnections between the processor and the memory subsystem, in some cases including external caches. Examples of such modules are MISRs attached to the bus, which update a signature every time new data are going to be written to the memory. This solution has been adopted by commercial microcontrollers, e.g., from Freescale [72]. In other cases, dedicated programmable embedded cores are in charge of tracing specific bus transactions (e.g., ARM [73]) and of storing a history of processor execution in a local memory, which is accessible through a dedicated port (e.g., for debug purposes). An example of IP core specially devised for SoC testing is described and demonstrated for a processor compliant with the SPARC v8 architecture in [74]. The presence of caches significantly limit the amount of data flowing through the bus, and hence the number of faults whose effects can be observed by observing it.

SBST programs using this observation method should include specific sequences of instructions that permit the propagation of the fault effects up to the system bus.

As an example, let us consider the faults affecting the circuitry that supports an external coprocessor. If the external coprocessor is connected to the processor with dedicated ports, the effects of such faults –propagated up to this interface in the original program– need to be read back from the coprocessor and stored to the system memory in order to become observable in the system bus. This solution adds complexity to the test program and is not always feasible. In case of faults whose effects can only be observed on non-functional output signals and never read back, the fault coverage reduction cannot be recovered, thus resulting in general in a potentially less-effective observation mechanism.

Also, if an SBST program developed for processor-level observation is evaluated resorting to this observation solution, a significant fault coverage reduction could be observed. This fault coverage loss is motivated by the reduction of the observed signals, as they are a subset of the output signals of the processor.

5.4 Memory Content Observation

According to this solution, a fault is marked as detected if the content of the system memory is different than the expected one at the end of the execution of the SBST program.

All the previously presented observation mechanisms rely on the fact that some output ports of the circuit can be constantly monitored, e.g., by a dedicated tester which is physically connected to test points or to the interface with on-board instruments. This is not the usual case of SBST in general. In a manufacturing at-speed SBST scenario, the functional program is often uploaded in the system memory (e.g., a cache, or a dedicated flash) and run at-speed, storing its responses in some available memory elements, such as internal registers, caches, or main memory, and hence permitting a low-cost tester to access them at the end of the execution. Similarly, during in-field SBST, at the end of the test program run the processor itself or another module (e.g., another processor) may perform an access to the specific memory cells in order to compare their values with the expected ones.

According to the presented scenario, this observation mechanism assumes that the test program collects in some way the information about test results and saves this information in the system memory. The information collected by the test program may be compacted by the test program itself and then (at the end of the test process) saved in few selected memory cells. Alternatively, the information saved by the program may be written to a set of memory cells, according to the targeted module characteristics as described in [75] for a MIPS-like processor and an industrial System-on-Chip.

Since the test results correspond to the values generated by the test program, which are checked only at the end of the test program execution without taking into account when these results are produced, some performance faults may escape when using this observation mechanism. For example, in the case of Branch Prediction Units, some performance faults may not modify the final test program results, but only delay the actual execution time [16], e.g., by turning a correctly predicted branch into a mispredicted one.

5.5 Performance Counters Observation

Performance Counters (PeCs) measure the number of occurrences of different internal events, making their observation easier from the outside. They exist in many processors, mainly for design validation, performance evaluation and to support silicon debug. Their values can normally be accessed via software. Hence, a test program may read the value of a given PeC, execute a sequence of instructions exercising a given module, and then read again the value of the PeC comparing it to the expected one. Possible differences may allow the detection of faults inside the module.

The most common types of PeCs include those that count internal events related to:

- caches, counting the number of miss and hit events;
- Branch Prediction Units (BPUs), counting the number of correctly or incorrectly predicted branches;
- pipeline stages, counting the different types of stalls;
- Memory Management Units (MMUs), counting the number of hit/miss accesses to the TLB;
- exception units, counting the number of triggered exceptions, often divided by type;

- bus interfaces, counting the number of performed bus transfers, also often divided by type.

These counters are already quite common in general-purpose high performance processors, and their adoption is growing in microcontrollers for embedded applications.

The usage of these counters as part of the observation mechanism adopted by a testing procedure was proposed in several works, such as [16] [14] that use variants of the MIPS architecture, or [76] [59] working with the OpenSPARC T1 processor. The PeCs have also been proposed as feedback in automatic test program generation [77] and in [54] to simplify the test programs aimed at detecting faults in caches. They are crucial for the detection of some specific types of faults, such as performance faults. Moreover, they can facilitate the test of faults belonging to some modules, such as Branch Prediction Units, Cache Controllers, TLBs. They may also be used to support the test of specific modules within the pipeline, such as those controlling the activation of stalls.

Regarding observability issues, the PeCs may provide deeper details on internal events affecting the module that may not reach the output ports, and allow the detection of several performance faults. Thus, exploitation of PeCs and propagation of performance values to system memory increases observability and may represent a valuable solution during in-field SBST.

5.6 Debug Interface Observation

The features currently provided by many processors in order to support the debug of the software can also be used for test purposes. Examples of such debug interfaces are the vendor independent standard Nexus IEEE-ISTO 5001 [78] [79] extensively used in U.S. automotive applications, and the ARM CoreSight On-chip Trace and Debug Architecture [80], widely adopted by various chip vendors in devices based on ARM Cortex-A, Cortex-M and Cortex-R cores.

These features often allow accessing to some information about the internal behavior of the processor during its normal operation (and without slowing it down). They typically allow tracing the sequence of instructions executed by the processor, either by writing them to ad hoc external interfaces that can be accessed on-the-fly or by storing them in a special memory buffer.

When trying to use these features to observe in-field test results, some difficulties arise: on one hand the mentioned special memory buffer usually has a reduced capacity, limiting its usage to very short test programs; on the other hand, the on-the-fly monitoring of the flow of data produced by the debug interface can only be done resorting to ad hoc hardware.

Although such ad hoc hardware is usually not available in a typical in-field test scenario, it may be added if some programmable hardware is available on the board or on-chip. This possibility becomes particularly attractive in the case of SoC platforms provided by FPGA vendors, often equipped with ARM processors and the abovementioned ARM CoreSight Architecture, i.e. the Zynq-7000 SoC platform by Xilinx or the SoC variants of the Cyclone V family by Altera. On these platforms, the ad hoc port provided by the debug interface is connected to the FPGA fabric, so that an on-the-fly monitor can be added with a moderate effort.

A scenario similar to the one described above was proposed in [81] [82], where other test oriented instrumentation is provided by a module mapped on an available on-board FPGA. A scheme for on-the-fly monitoring of the execution trace data is presented in [83] and its effectiveness to detect control flow errors, i.e. faults that modify the normal program execution flow, is experimentally evaluated via fault injection on miniMIPS and LEON3 systems.

To better understand the possible uses of the debug features for test purposes, a brief description of the ARM CoreSight Architecture is included here.

According to the CoreSight Architecture, every configurable CoreSight component contains a set of memory-mapped registers that are accessible from external debug equipment and from the application software. The CoreSight components can be classified in:

- *Access and Control* components. They allow to access memory and memory mapped registers, including the configuration registers of the CoreSight system. The ARM *Debug Access Port* (DAP) provides a bridge between a reliable low pin count external interface referred as the *Debug Port* (DP) and on-chip memory and memory-mapped registers through different *access ports* (AP) connected to the system buses. The Debug Port can be for example a JTAG port. Each Access Port implements a master port that interfaces to one of the standard memory-mapped interfaces, such as APB (ARM Peripheral Bus), AHB (Advanced High-performance Bus) and AXI (Advanced eXtensible Interface).

- *Trace source* components. Trace information is generated by different components that are masters of the *AMBA trace bus* (ATB). The *Program* (PTM) [84] and *Embedded* (ETM) *Trace Macrocells* generate trace data containing information of the software running on the processor. The *Instrumentation* (ITM) and *System* (STM) *Trace Macrocells* allow the software developer to explicitly insert trace points into the software to ease application-level trace and debug (e.g., print type debug). The *AHB Trace Macrocell* (HTM) provides tracing of AHB buses. The *Fabric Trace Monitor* (FTM) is a Xilinx specific trace Macrocell that complies with the CoreSight architecture specification and enables to trace data generated inside the programmable logic (FPGA) of the Zynq-7000 SoC.

- *Trace link* components. The interconnection components include the *ATB Funnel* to merge trace data from multiple sources (PTM, FTM and ITM) into a single stream driving an ATB bus, and the *ATB Replicator* that duplicates the trace stream onto two output ATB master ports, which can be connected to trace sink components.

- *Trace sink* components. There are two types of trace sink components in the Zynq-7000 device. The *Embedded Trace Buffer* (ETB) is an on-chip storage module with limited size (4KB) which enables short-window real-time and full-speed tracing. The *Trace Port Interface Unit* (TPIU) allows the trace packages to be output to the FPGA part or to chip output pins for on-the-fly processing.

The block diagram of a CoreSight System typical implementation is shown in Figure 5.2.

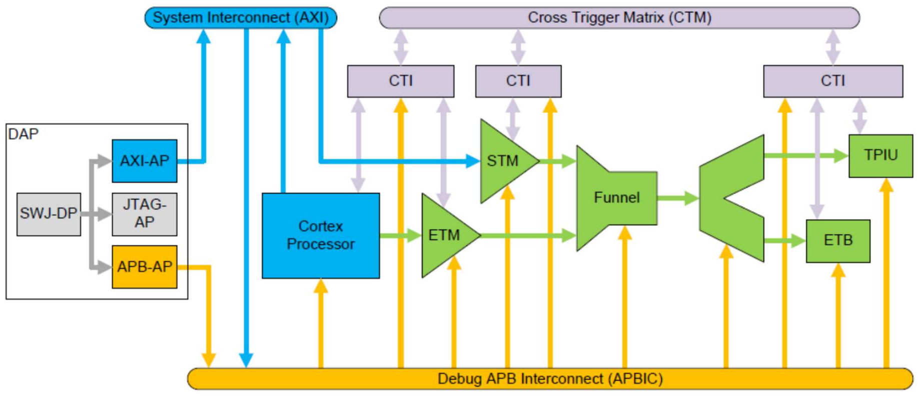

System Interconnect (AXI)
Cross Trigger Matrix (CTM)
CTI
CTI
CTI
DAP
AXI-AP
SWJ-DP
JTAG-AP
APB-AP
Cortex Processor
ETM
STM
Funnel
TPIU
ETB
Debug APB Interconnect (APBIC)

Figure 5.2: Example CoreSight system (source [85]).

The trace source components like the Program Trace Macrocell can report several relevant information about the test program execution, including, for example, taken/not-taken branch decisions, the target PC address of branch instructions, cycle accurate information between two branch instructions and the exception status of the processor [84].

Chapter 6

Observation Techniques – Experimental Results

In this section, we present some experimental figures aimed at assessing the fault detection abilities of the different observation mechanisms described in the previous section. The observation solutions are denoted as follows:

- S1: module-level observation
- S2: processor-level observation
- S3: system bus observation
- S4: memory content observation
- S5: performance counters observation
- S6: debug interface observation

Three test cases were considered, using different processor-based systems: a single core MIPS-like processor system (Test cases #1 and #3) and a multicore Leon3 processor system (Test case #2). For two of the test cases (#1 and #2) a module was selected within the processor and a test program was produced, targeting the faults inside the selected module. The selected modules were the Branch Prediction Unit in test case #1 and the data cache controller in test case #2. They were chosen because they are known to produce performance faults, a kind of fault that is hard to detect using the traditional memory content observation method. For test case #3 the test program targets the faults inside all the internal modules of the processor.

Observation solution S6 was considered only on the test cases based on the MIPS-like processor (Test Cases #1 and #3). Also, in these two test cases a second performance counter mechanism consisting of a simple timer was considered, denoted as S5*.

For every considered observation solution (S1-S5 in Test Case #2, S1-S6 in the others), a fault simulation campaign was run in order to determine whether a fault produce a difference in the observed outputs if compared with the golden run (i.e., the simulation of the system in a fault free condition). A commercial fault simulator was carefully setup in order to mimic the described observation solutions targeting the stuck-at faults in the selected modules. In some cases, it was necessary to introduce some slight modifications to the system, oriented to replicate during the fault simulation campaign the observation methods employed in every experiment. In the fault simulation experiments performed for a given test case, the same test program was used for all the different observation methods.

6.1 Test Case #1: Branch Prediction Unit

For this test case we considered a MIPS-like processor based on the RT-level VHDL description available at [86]. This processor is a 32 bits core composed of:

- a pipeline with 5 stages: address calculation (PF), instruction extraction (EI), instruction decoding (DI), execution (EX) and memory access (MEM);
- a System Coprocessor, responsible for the management of interrupts and exceptions;
- a Register Forwarding and Pipeline Interlocking unit, dealing with data hazards among the pipeline stages;
- a Branch Prediction unit (BPU), implementing a Branch Target Buffer;
- a Register Bank, consisting of thirty two 32-bit registers.

In this case we considered the faults belonging to the Branch Prediction Unit (BPU), which is a commonly used solution for decreasing the negative impact of branches on the performance of pipelined architectures. One of the most widely used implementations of a BPU is based on the Branch Target Buffer (BTB), i.e., a data structure consisting of a set of entries, each one containing:

- the address of a branch instruction;
- the expected target address in case the branch is predicted as taken;
- possibly, one or more bits, storing the prediction (taken, or not taken).

During the instruction fetch cycle and providing that the instruction is a branch instruction, the processor core is able to anticipate the branch outcome by accessing the BTB. In case of a correct prediction, the processor can fetch the correct instruction during the next clock cycle. Figure 6.1 shows the BPU inputs and outputs and their connections to the related pipeline stages of the MIPS-like processor.

This module is particularly interesting, due to the fact that faults affecting it do not always cause the generation of erroneous results, but often impact the processor behavior by simply slowing down the system, i.e., increasing the number of mispredictions and possibly causing the system not to match the expected target in terms of performance. Such faults are also referred to as performance faults and can be tested by using some properly devised performance counters.

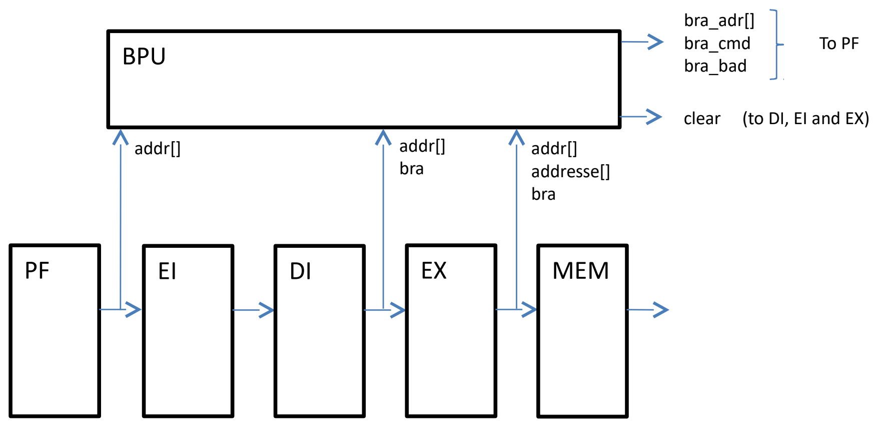


Figure 6.1: Branch Prediction unit and related pipeline stages of the MIPS-like processor.

6.1.1 System setup

Minor changes were done to the processor design, mainly to correct a bug affecting the Branch Prediction Unit. The VHDL description was synthesized using the Synopsys Design Compiler with a technology library developed in-house. Both the technology library development and the changes to the processor design were available at the beginning of the present work. The complete processor area is 41 959 equivalent gates and 2 112 equivalent D flip flops, and is described in 3 131 lines of VHDL code, while the BPU accounts for 4 248 equivalent gates, 283 equivalent D flip flops and 274 VHDL lines. The considered fault list consists of the single stuck-at faults inside the BPU. It has a total of 27 354 faults, corresponding to about 10% of the stuck-at faults inside the whole processor (268 424 faults). No undetectable faults were marked as such by the fault simulation tool.

With the purpose of testing the Branch Prediction Unit, a functional test program was manually developed following the algorithm described in [14]; it occupies 308 bytes of ROM memory, and its execution requires 5 229 clock cycles. The program is a sequence of taken and not taken branches, suitably crafted to exercise each entry of the BTB memory, and each bit of the associated comparator used to determine if the address of a branch instruction being processed matches with the one stored in the table entry. A signature that compacts the sequence of the executed branch addresses is produced and written to memory at the end of the test. Using this test program, we ran several fault simulation experiments aimed at assessing the effectiveness of the different observation mechanisms when addressing single stuck-at faults. Fault simulation experiments were performed using Synopsys TetraMAX, which is a well-known tool used for manufacturing-testing-oriented fault simulation. A proper framework based on TetraMAX was devised in order to evaluate the considered observation solutions.

Figure 6.2 shows the interface of the MIPS-like processor. In order to con-

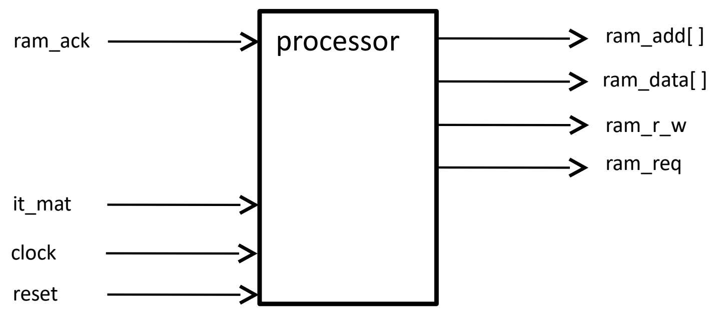


Figure 6.2: MIPS-like processor external interface.

sider a realistic scenario, the synthesized processor was embedded together with a RAM memory and a ROM memory containing the program code. For the sake of simplicity, no peripheral cores were included in the system, thus no external interrupts could have been raised during the simulation experiments.

More in details, a proper module wrapping the processor was added to the system and used as the top module in TetraMAX fault simulations. This wrapper has a double purpose: on one hand, it contains all the elements required by the different observation mechanisms, while exporting the different sets of observed signals as primary outputs of the wrapper as described below. On the other hand, it allows the inclusion of the program memory inside the system that is fault simulated by TetraMAX, hence giving a much closer match between the simulated and real behaviors in the presence of faults.

The wrapper includes a RAM memory used by the Memory Content observation mechanism and additional logic aimed at implementing the Performance Counters and Debug Interface observation mechanisms described in the previous section. Two different performance counters are implemented: the first one obtains the test duration by capturing the value of a simple timer at the end of the test program execution; the second one is able to count the occurrences of incorrectly predicted branches by means of one of the outputs of the Branch Prediction Unit (the *clear* signal shown in Figure 6.1) which is activated each time a wrong prediction is detected. In order to emulate a solution based on debug interface observation, a set of internal signals produced by the execution stage of the pipeline were exported up to the wrapper interface as described below.

The wrapper is shown in Figure 6.3 and has the following characteristics:

- Two blocks of memory are used; the ROM memory contains the test program code, while the RAM memory is initialized before the simulation and receives the test program results at the end of execution;
- the clock and reset primary input signals are applied to the processor by the wrapper; at the system reset, the value of the processor's Program Counter

corresponds to the first available address in the ROM memory;

- a second read port was added to the RAM memory to support the memory content observation mechanism. Using the additional address inputs (*spy_addr*) and data outputs (*spy_dout*), the memory content is observed without interfering with the normal memory access operations performed by the test program;

- internal signals were exported up to the wrapper boundary in order to support module level and debug interface observation mechanisms;

- a timer register and a counter were added to emulate the two performance counter observation methods considered. The timer register is enabled at the end of the test to capture the value of a timer input *t_in* provided by the testbench; the *Perf_cnt* counter increments each time an incorrect branch prediction arises.

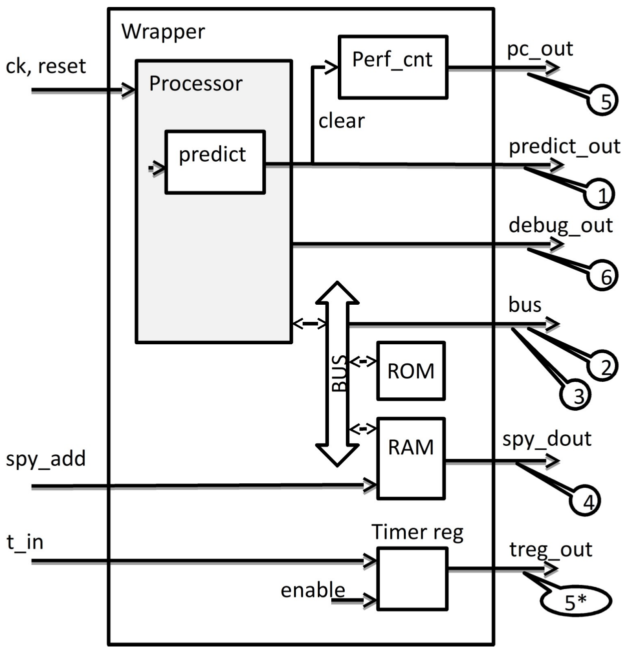


Figure 6.3: BPU's test case: experimental environment and considered observation solutions.

The fault simulation experiments were driven by a Value Change Dump (VCD) file produced via logic simulation by means of a test-bench surrounding the wrapper, which represents the top module in the TetraMAX simulations. The VCD file provides the proper stimuli to the wrapper's primary inputs: *ck*, *reset*, *spy_addr* (to select the RAM location observed at *spy_dout*) and *t_in* (the timer signal provided by the testbench). In detail, the reset signal was only triggered as soon as the simulation was started and then remained inactive until the end. At the end of the test program execution, the *spy_addr* value continuously scans the memory interval between the first and the last address of the RAM zone to be observed through *spy_dout*. The enable signal of the Timer register is decoded from a memory write operation added at the end of the test program in order to capture the value of time at the end of execution.

During the fault simulation campaigns, TetraMAX was provided with the same fault list for all of the performed experiments (consisting of all the stuck-at faults on the BPU). The different observation solutions (also shown in Figure 6.3) are described below. It must be remembered here that the first three solutions are usually not feasible in an in-field scenario. Solutions S4 to S6 can be implemented in-field by a few instructions at the end of the test program that read the final value of the observed magnitude and compare the read value with the expected one. Instead, in order to assess the fault coverage using a fault simulation tool like TetraMAX, the final value of the observed signal must be presented as a primary output of the top entity being simulated, i.e. the wrapper, and the fault simulator must be instructed to only observe this primary output, and to observe it only at the end of the experiment. Each solution was implemented as follows:

- S1: the primary outputs of the Branch Prediction module (the predicted address, and other few control signals shown in detail in Figure 6.1 and grouped as *predict_out* in Figure 6.3) are observed during the whole simulation. To make this possible, the processor interface was modified to export the outputs of the internal Branch Prediction module up to the wrapper interface.

- S2: the sub-set of signals composed of all the original processor outputs is observed during the whole simulation (the memory data, address and control interface signals detailed in Figure 6.2 and labeled as *bus* in Figure 6.3).

- S3: the original processor outputs related to the system bus are observed during the whole simulation. This solution is identical to S2 in this case study because all the outputs of the MIPS-like processor are system bus related.

- S4: the content of the memory area where the test program is expected to write is observed at the end of the test program execution; this is implemented using the wrapper. More in details, the fault simulation experiment is set up so that upon the test program completion, the observation is enabled on the outputs *spy_dout*, while the inputs *spy_addr* sweep the RAM memory interval devoted to save the program results. Two words are enough for the present example because a signature is used to compact the test program results.

- S5: incorrectly predicted branches performance counter. Upon the test program completion, the TetraMAX fault simulation experiment enables the observation of the primary outputs corresponding to the Performance Counter value (*pc_out* in Figure 6.3).

- S5*: timer performance counter. In a way similar to the previous solution, upon the test program completion the observation of the primary outputs corresponding to the Timer register is enabled (*treg_out* in Figure 6.3).
- S6: a set of internal signals produced by the execution stage of the pipeline is exported up to the wrapper interface. The exported signals are the address (*instr_adr*) and opcode (*instr*) of the instruction being executed, along with two control signals indicating that an instruction is effectively being executed (*instr_adr_valid*) and that it is a branch instruction (*instr_bra*). The set of signals is grouped as *debug_out* in Figure 6.3, and the observation is enabled only at the instants that a branch instruction is executed in the non-faulty execution run (golden run).

In the case of solution S6, the aim is to emulate the observation of a signature that compresses the sequence of branch instruction information on each branch execution. In the real system the signature can be computed by ad-hoc hardware as described in section 5.6 *"Debug Interface Observation"*. Note that the observation instants are fixed according to the golden run timing, independently of the actual timing of the faulty execution. So, if the fault produces timing modification only, i.e. if it is a performance fault, in most of the cases the observed signal values at the fixed observation instants will differ from the golden run values, and hence the fault is detected by the test. In other words, the experiment results ressemble the results of a real system in which the abovementioned signature includes not only addresses and taken/not taken information, but also a timestamp.

6.1.2 Results

Class	S1	S2/S3	S4	S5	S4+S5	S5*/ (S4+S5*)	S6/ (S4+S6)
Detected	21 631	21 614	11 259	19 902	19 902	19 861	20 201
Not Detected	5 723	5 740	16 095	7 452	7 452	7 493	7 153
fc [%]	79.08	79.02	41.16	72.76	72.76	72.61	73.85
fc/fc(S1) [%]	100.00	99.92	52.05	92.01	92.01	91.82	93.39

Table 6.1: BPU's test case: fault simulation results

The fault simulation results for the Branch Prediction Unit of the MIPS-like processor are reported in Table 6.1. As explained above, for this processor the observed signals in mechanisms S2 and S3 are exactly the same. For this reason a single column labeled as S2/S3 is shown in Table 6.1. About 79% of the total faults in the module were detected by exploiting the observation mechanism S1:

this represents the maximum achievable fault coverage, given the test program we considered. Only a few of these faults were not propagated through the system bus (S2/S3). This happens because the test program used in the experiments was carefully developed focusing on the end-of-manufacturing scenario; thus, a small minority of the faults detected by S1 were marked as not observed by TetraMAX in the S2/S3 scenario.

On the contrary, a significant fault coverage drop can be noticed when moving from S2/S3 to S4, when the content of the main memory is observed at the end of the test program execution. In this case, only 41% of faults are marked as detected, meaning that about a half of the faults that were propagated by the test program to the system bus (experiment S2/S3) were not consistently saved in the main memory, turning them into not observable in a typical in-field scenario. As described in the introduction, part of these faults are masked, however, those of them that are not performance faults may become observable if correctly propagated up to the system memory. Thus, this result could be improved by the test engineer in charge of test program, if this scenario had to be targeted.

The interesting result concerns the experiment which used the Performance Counter (S5): in this case we have about 73% of detected faults, meaning that only 6% of faults were not observable with respect to the results of processor-level observation (S2/S3). Since the Performance Counter is tightly related to the module under consideration, its effectiveness was very high, providing a good example of how resources existing in real processors can be exploited for in-field testing. This result also suggests that in the present case, performance faults are by far the most important factor contributing to the coverage reduction from S2/S3 to S4.

Similar good results were obtained for the Timer (S5*) and Debug Interface (S6) solutions, being S6 slightly better. As in the case of solution S5, the information provided by the Debug Interface (the time and address of each branch instruction) is also tightly related to the Branch Prediction module. The results also show that the Timer is a good alternative to observe the faults escaping S4, which seems reasonable as most of these faults are presumably performance faults.

Further analysis on the fault lists of the different experiments showed that all the faults that were detected for S4 were also included in the detected fault list of S5, S5* and S6. Consequently the results of combining S4 and each one of the other three solutions are identical to the results obtained with only the other solution. For example, the results of solution S5 and the combination of solutions S4+S5 are the same. That is the reason for the double label (for example S6/(S4+S6)) in the rightmost columns in Table 6.1. As discussed above, the Performance Counters can usually be accessed as a system peripheral, thus its values can be read by the test program itself at the end of its execution, and then used to update the test signature. Therefore, in a typical in-field scenario, the adoption of solutions like S5 or S5* allows achieving high fault coverage, with the minor additional effort of the Performance Counters reset and reading operations.

6.2 Test Case #2: Data Cache Controller

The second test case corresponds to a multi-core system based on the LEON3 processor, whose synthesizable VHDL model is freely-available for research purposes under GNU GPL license at [64], as part of the GRLIB IP library [87]. The LEON3 processor is a highly configurable 32-bit core compliant with the SPARC V8 architecture and is composed of:

- a pipeline with 7 stages;
- AMBA-2.0 AHB bus interface;
- separate instruction and data caches, which can be configured in several ways;
- several optional components: high-performance, fully pipelined IEEE-754 Floating-Point Unit; Memory Management Unit; Debug Support Unit with instruction and data trace buffer.

LEON3 data cache can include data cache coherency features based on the snooping technique. Data cache snooping is of high importance for multiprocessor systems. The purpose of this logic is to keep the data cache synchronized with external memory and other caches, avoiding any data inconsistency. This task is accomplished by monitoring the write accesses on the AMBA AHB bus to cacheable memory locations: if another AHB master writes to a cacheable location that is currently cached in the data cache, the corresponding cache line is marked as invalid.

In this case, the considered faults where those belonging to the controller module of the data cache. As in test case #1, some of the faults inside the selected module may cause the processor to produce correct results but delayed in time, i.e. they are performance faults.

6.2.1 System setup

The GRLIB IP library provided by Cobham Gaisler AB was exploited in order to create the Symmetric Multi-Processor (SMP) embedded platform depicted in Figure 6.4. Note that this platform permits to exercise the data cache coherency logic. Briefly, the system included:

- two LEON3 processors, referred to as *core0* and *core1* in the figure, each one attached to the AMBA-2.0 AHB bus as master;
- a bank of SDRAM memory, and a ROM memory loaded with the program;
- an interrupt controller;

- a 16-bit GPIO port.

Both instruction and data caches in LEON3 processors were configured as 1-way, 1 Kbyte, 16 bytes/line. The data cache policy is always write-through, with no cache allocation on a write miss. The snoop mechanism was enabled to assure cache coherency.

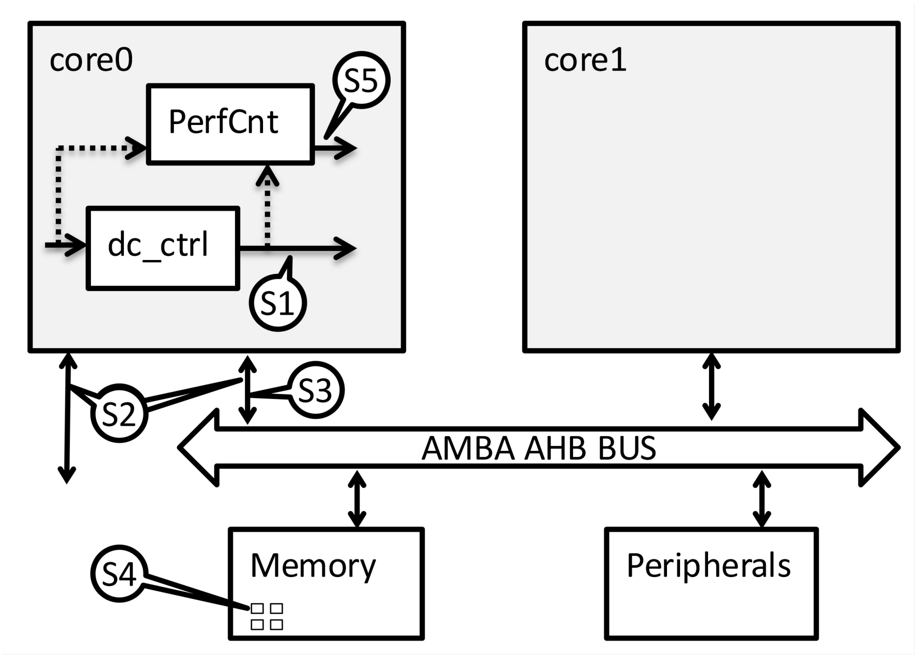


Figure 6.4: Data Cache Controller test case: dual-core system under test and considered observation solutions.

The whole two core system was simulated without faults to obtain the patterns applied to the system during fault simulation. The internal module under analysis was in all the cases the data cache controller of *core0* (*dc_ctrl* in Figure 6.4), and the considered faults were the stuck-at faults inside it. In order to minimize computation effort, only part of the system was included in each fault simulation experiment. For solution S1, the top module of the system in the fault simulation experiments was the data cache controller of *core0*, denoted as *dc_ctrl* in Figure 6.4. For solution S5 it was also included the performance counters module described below, while for the rest of the observation mechanisms the fault simulated system includes the complete *core0* processor.

In order to better assess the observation solution based on performance counters, the system was enriched by adding some of the performance counters usually available on commercial processors. For this purpose, a new block was designed, called *PerfCnt* in Figure 6.4, fed by a subset of the data cache controller ports. The set of performance counters implemented by the block can be classified in the following two groups:

1. *AMBA bus events*: a group of 5 counters triggered on the occurrence of specific bus transfers (write, read, byte, half word, and word transfers). These counters take their inputs from the connections between the data cache controller and the AMBA AHB interface.

2. *Cache operation events*: 3 counters for cache read hit, cache write, and cache line invalidation because of snoop events. The inputs of these counters come from signals connecting the data cache controller with the cache memory and the integer unit.

The whole system was synthesized with Synopsys Design Compiler using the Synopsys SAED32 standard cells library [88], one of the libraries provided in the synthesis toolkit of the processor. The complete Leon3 processor core is described in 9 578 VHDL code lines. Each of the two instances of the Leon3 processor core occupies 155 568 equivalent gates, and 23 784 equivalent D flip flops. As mentioned before, the module under analysis is the data cache controller in *core0* processor, which is described in 1 669 VHDL lines, and occupies 6 380 equivalent gates and 339 equivalent D flip flops. The fault list used during fault simulation experiments consists of all the possible single stuck-at faults affecting the module under analysis, which totalize 23 958 faults, a fraction of the 715 838 stuck-at faults of the whole processor core.

For the purpose of this set of experiments, a test program was created based on the techniques described in [13] and [25], aimed at detecting faults affecting different specific functionalities inside the data cache controller. One of these functionalities is the snooping logic, whose basic function is to invalidate a given cache line when another processor executes a write operation on the memory block it stores. The part of the test that targets the snooping logic runs concurrently on both processors, as described in [25] and in 4.3 *"Proposed approach"*. It consists of a sequence of memory accesses exciting the inputs of the address comparators used to detect the cache line invalidation condition and verifying whether the corresponding validity bit is modified accordingly. The rest of the test program runs only on *core0*. It includes a section targeted to the replacement logic [13] and the programs provided by Cobham Gaisler AB to verify the functionalities of the cache controller. The test program was written in C language except for some small sections which are written at the assembly level; its execution time is about 300k clock cycles, and its code occupies about 26 000 bytes.

To allow a fair comparison, the test program was exactly the same in all the experiments. Each time, a sub-set of the outputs was selected and made observable by TetraMAX in order to mimic the different observation methods presented in chapter 5 *"Observation Techniques – Survey"*. As mentioned above, the top module in the fault simulation runs was the data cache controller inside *core0* (*dc_ctrl*) for solution S1, the same plus the performance counters block for solution S5, and the *core0* processor in the rest of the cases. For each of the considered solutions, the sub-set of observed signals is listed below, as well as the observation points (also shown in Figure 6.4):

- S1: the sub-set of primary outputs corresponding to all the output ports of the data cache controller (*dc_ctrl* module) is observed during the whole test program execution.
- S2: the sub-set related to all the outputs of the *core0* processor is observed during the whole test program execution. This sub-set includes the AMBA AHB bus interface outputs described in the next paragraph, the outputs to an external debug support unit (DSU3) and the outputs to an external interrupt controller (IRQMP) that also manages the startup sequence of the multiple processors.
- S3: the sub-set related to the outputs of the *core0* processor that are connected to the AMBA AHB bus is observed during the whole test program execution. It includes the 32 bit address bus, the 32 bit data bus and the full set of control signals of an AMBA AHB master interface.
- S4: the final signature computed by the test program out of the data written in memory is observed at the end of the test program execution.
- S5: the primary outputs corresponding to the PerfCnt module are observed at the end of the test program execution (outputs of the eight counters described above).

A problem arises with solution S4. The memory was left outside the model of the system provided to the fault simulator, and consequently we cannot observe the final memory value. Instead what was done was to observe the data bus in the moments a write transfer is done at the AMBA bus. As explained in subsection 6.5.1 *"Setting the fault simulator observation times"*, this approach is not totally correct and may produce optimistic results.

6.2.2 Results

Table 6.2: Data Cache Controller's test case: fault simulation results

Class	S1	S2	S3	S4	S5	S4+S5
Detected (DT)	12 112	8 517	8 516	7 511	2 479	7 796
Not Detected (ND)	9 726	9 627	9 628	10 633	17 483	17 483
Undetectable (UD)	2 120	5 814	5 814	5 814	3 996	3 996
fc [%]	50.56	35.55	35.55	31.35	10.35	32.54
fc/fc(S1) [%]	100.00	68.07	68.06	60.03	19.81	64.37

The fault simulation results for the data cache controller module of the LEON3 processor are summarized in Table 6.2. The gathered results show, as expected, quite different fault coverage figures when the different observation mechanisms are adopted. The limited fault coverage achieved by the different solutions (including

S1) is mainly due to the high amount of redundancy (hence, untestability) existing in the considered implementation of the Data Cache Controller module, which could not be removed by the synthesis tool. This redundancy stems from the high flexibility of the LEON3 cache system, which can be configured at compile time to implement different cache solutions, not only in terms of cache size, but also of cache replacement and writing mechanism (e.g., write-back or write-through). Unfortunately, after configuration the VHDL code contains unused structures that are not always removed by the synthesis tool, leading to redundancy. TetraMAX is partially able to identify untestable faults and classify them as Undetectable (UD) under different situations: during the net-list compile, because they are located on circuitry with no connectivity to an externally observable point, or because the fault effect is blocked from propagating to an observable point due to tied logic; alternatively, during the fault simulation, when some faults are proved to be untestable since they are located on redundant circuitry, by means of formal analysis. However, part of the faults, which cannot be tested in a functional manner, escape from the analysis of testability made by TetraMAX, and simply result as part of the Not Detected (ND) set.

In order to consider that there is a high percentage of untestable faults, the bottom row in Table 6.2 shows the fault coverages expressed as a percentage of the one obtained with solution S1. It can be seen that the number of faults detected by solutions S2 and S3 is almost 70% of the one detected by solution S1, meaning that a significant amount of faults that are observable at module-level are not propagated up to the processor output signals.

Interestingly, solution S2 covers only one fault more than solution S3. This was partly expected, because the data cache controller module is strongly memory related. A more appreciable coverage difference between solutions S2 and S3 can be expected for a non-memory related module. Such a difference may be produced because the fault effects are not properly propagated to the system bus by the test program (e.g. a coprocessor module), or because the fault only affects resources that cannot be read back in a controlled way by the test program (e.g., an interrupt controller).

The provided results also show that the number of faults detected by S4 is about 88% of the number of faults covered by S3, and the number of faults covered by S5 is only one third of the number of faults covered by S4. However, it is important to analyze the intersections and differences between the sets of faults detected by S4 and S5.

As mentioned in the previous sections, in an in-field SBST approach solutions S4 and S5 are the only practically feasible ones, since they do not require the use of any additional hardware devices. Figure 6.5 schematically shows the effect of adding the results obtained by S4 and S5. S5 provides 285 additional detected faults, about 3.8% of the number of faults detected by S4 alone. S4 and S5 combined detect 91.5% of the faults covered by S3. Remarkably, there are some faults (104) covered by S5 and S3 that are not detected by solution S4; these faults are associated to the group of performance counters related to AMBA bus events

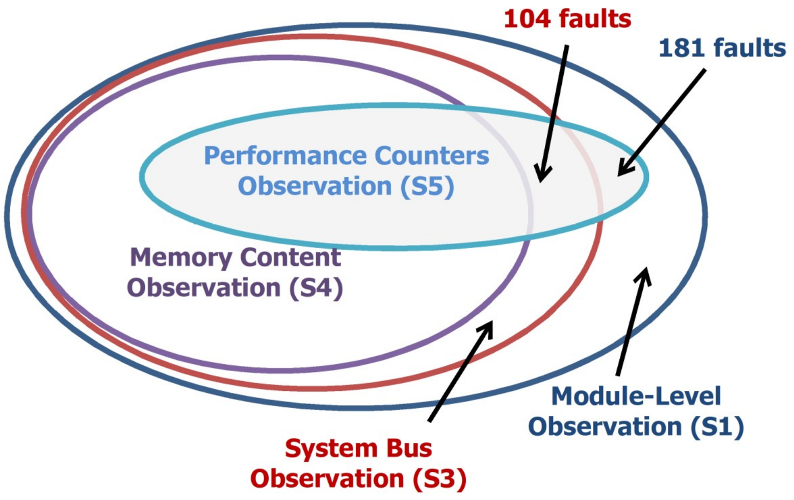

104 faults
181 faults
Performance Counters Observation (S5)
Memory Content Observation (S4)
Module-Level Observation (S1)
System Bus Observation (S3)

Figure 6.5: Data Cache Controller's test case: sets of faults detected by the different observation solutions.

(see the previous sub-section). This group of faults shows a contribution of S5 which partly compensates the loss in fault coverage moving from S3 to S4.

Additionally, there is another set of faults (181 faults) covered only by S5 that escape the test not only by S4 but also by S3. This means that during the execution of the test program the effect of these faults is unobservable at the bus level, but modifies the behavior of the second group of performance counters (cache operation events, described in the previous sub-section). The considered test program is successful in exciting these faults but is not able to produce observable modifications at the bus level. The observation of the performance counters contributes to the overall coverage by making these faults observable.

6.3 Analysis of the Results of Test Cases #1 and #2

Experimental results successfully confirmed some a-priori considerations on the implemented observation solutions. Considering solutions S1 to S5, it can be stated by construction that, for a given test program, the higher coverage is the one obtained resorting to solution S1. Module level observation S1 offers the highest observability even if it can hardly be implemented in a real scenario. For all the other solutions except S6, the observed signals are a subset or a transformation of the signals observed in S1. So, for example in test case #2, if the effect of a fault inside the *dc_ctrl* module is observable at the AMBA bus level (S3), it can also be observed on the *dc_ctrl* ports (S1).

In a similar way, the following inclusion relationships (also summarized in Figure 6.5) can be assured between the sets of detected faults and between the sets of undetectable faults in the different solutions. The name in parentheses identifies the solution and the two letter code denotes the fault class (DT: detected, UD: undetectable).

$$DT(S1) \supseteq DT(S2) \supseteq DT(S3) \supseteq DT(S4) \tag{6.1}$$
$$UD(S1) \subseteq UD(S2) \subseteq UD(S3) \subseteq UD(S4) \tag{6.2}$$
$$DT(S1) \supseteq DT(S5) \tag{6.3}$$
$$UD(S1) \subseteq UD(S5) \tag{6.4}$$

All these relationships were verified in both test cases by analyzing the detailed fault lists for each of the observation solutions.

An additional experiment was performed in test case #2 trying to understand how much the final fault coverage can be enhanced if a larger set of performance counters is used. In this experiment, the data cache controller subset of signals used as inputs by the performance counters module was observed during the whole experiment. Denoting by S* this new observation mechanism, the following extensions to the above inclusion relationships can be stated:

$$DT(S1) \supseteq DT(S*) \supseteq DT(S5) \tag{6.5}$$
$$UD(S1) \subseteq UD(S*) \subseteq UD(S5) \tag{6.6}$$

In other words, all the faults detected by S5 are also detected by S*, and this remains true even if the internal design of the *PerfCnt* module is modified, as long as it has the same inputs. So, the coverage obtained by solution S* is an upper bound to the coverage that can be obtained with the same test program by any design of the *PerfCnt* module with the same inputs. The results for test case #2 show that solution S* adds 511 new detected faults with respect to S4. Another interesting result from this new fault simulation experiment is that, when observing the performance counters inputs, the set of undetectable faults is identical to the one obtained with S1. This means that the same faults detected by S1 could be potentially detected by adding new Performance Counters and devising a suitable input sequence.

Finally, it is interesting to compare the coverage results of combining S4 and S5 in both test cases. In the Branch Prediction Unit case there are two points to highlight: S5 provides a very good coverage (92% of the coverage obtained by S1) and the set of faults covered by S4 is completely included in the set of faults covered by S5 as shown in Figure 6.6. This is explained because the misbehaviors produced by the faults inside the Branch Prediction Unit naturally affect performance, and modify the count of incorrectly predicted branches, but most of them do not affect the result of calculations and therefore they do not affect the final signature value.

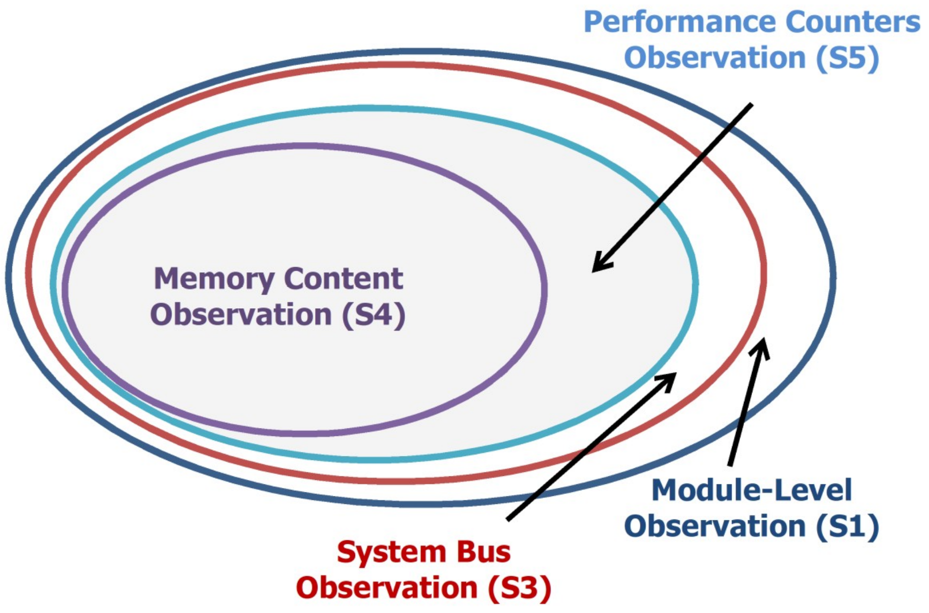

Performance Counters Observation (S5)
Memory Content Observation (S4)
Module-Level Observation (S1)
System Bus Observation (S3)

Figure 6.6: Branch Prediction Unit's test case #1: sets of faults detected by the different observation solutions.

On the other hand, for the Data Cache Controller test case, the performance counters observation was introduced aiming to cover the faults that turn a hit into a miss, or vice versa. The relatively low coverage obtained in solution S5 suggests that, in the complete cache controller, an important amount of faults exist that do not produce this kind of hit/miss permutation, and therefore modify neither the count of AMBA bus transfers, nor the count of hits/misses. Fortunately, these faults affect the processed results and can be exposed by a proper signature as shown by S4. Interestingly, by exploiting contemporarily solutions S4 and S5 (rightmost column in Table 6.2), the obtained results are similar to those obtained by observing the processor's primary outputs as in one of the end-of-manufacturing testing scenarios.

6.4 Test Case #3: Full MIPS-like processor

An additional test case was developed with the goal of evaluating how the chosen observation mechanism solution differently affects each of the internal processor modules. Also, the attention is focused on showing some figures to quantitatively assess their presence in the different parts of a pipelined CPU module. The reported analysis can be precious for the test engineer to cleverly decide which observation mechanisms have to be implemented, and for the designer of a CPU or SoC to judge whether it is worth to add some special hardware to support SBST code development.

The test case was initially developed to evaluate an existing observation solution based on debug interface observation and was later enriched with additional observation methods. The preliminary results were presented together with the debug interface observation solution in [27]. This observation solution consists of an on-the-fly monitor developed by Boyang Du, which exploits the already introduced ARM CoreSight Architecture [89] features that are available in the Zynq-7000 SoC platform by Xilinx [90]. The monitor is implemented in the FPGA part of the Xilinx chip. It observes the output of CoreSight's Trace Port Interface Unit (TPIU) and produces a compressed signature of the branch execution trace; the signature can be read at the end of the test program execution.

As a gate level model of the ARM hard core processor inside the Xilinx chip is not publicly available, it is not possible to evaluate the fault coverage obtained by such a scheme in the Zynq-700 SoC platform. To assess the effectiveness of this observation method, the already presented MIPS-like processor was used in an experimental setup identical to the one described in Test Case #1. As mentioned above, the Debug Interface observation solution was first introduced for Test Case #3. To do so, proper outputs were added to mimic the information available at the Trace Port Interface Unit in the original CoreSight system in the Zynq-700 SoC platform. Afterwards, the experiments for this observation solution were replicated for Test Case #1.

6.4.1 System setup

As in the other test cases, Synopsys TetraMAX was used to perform the fault simulation experiments to assess the fault coverage achievable with the different observation mechanisms.

The main differences between Test Cases #3 and #1 are two. First, in Test Case #3 the target faults are the stuck-at faults in the whole processor (268 424 faults), not only the faults inside the Branch Prediction module. Second, accordingly a different test program was used.

The test program used in Test Case #3 was manually developed by a test engineer knowing the netlist of the processor, targeting the maximization of the stuck-at fault coverage for the whole processor when using processor level observation (S2). Some quick modifications were done to assure that the memory write operations included in the test program are accessing RAM space and not ROM or unused parts of the memory space, so that a proper comparison can be done with the solution based on observing the final memory contents (S4). The test program was written in assembly language. The size and duration of this test program are 1 576 bytes and 19 298 clock cycles, respectively.

The observation mechanisms were identical to the ones described in Test Case #1. Solution S1 (module-level observation) is less meaningful in the present test case due to the fact that the test program was developed to obtain a good coverage in the whole processor, not only in the Branch Prediction Unit as was the case in Test Case #1. The observation mechanisms were described in detail for Test Case #1 and are summarized below for clarity.

- S1: module-level observation. Branch Prediction Unit outputs.
- S2/S3: processor/system bus observation. Identical in the present test case.
- S4: memory content observation. Memory content at the end of test.
- S5: performance counters observation. Branch prediction unit related performance counters observed at the end of test.
- S5*: Timer. Test duration obtained by triggering a timer register at the end of the test.
- S6: debug interface observation.

6.4.2 Results

Table 6.3 reports the number of detected (DT), possibly detected (PT), not detected (ND) and undetectable (UD) faults (out of the total of 268 424 faults) identified using each of the different observation mechanisms.

Table 6.3: Full MIPS-like processor test case: fault simulation results

	S1	**S2/S3**	**S4**	**S5**	**S5***	**S6**
DT	48 964	240 471	231 895	46 688	46 089	49 161
PT	152	995	1 953	1 104	1 104	180
ND	216 879	24 529	32 147	218 203	218 802	216 654
UD	2 429	2 429	2 429	2 429	2 429	2 429
total	268 424	268 424	268 424	268 424	268 424	268 424
tc	18.41 %	**90.40 %**	**87.18 %**	17.55 %	17.33 %	18.48 %
fc	18.24 %	89.59 %	86.39 %	17.39 %	17.17 %	18.31 %

It is worth noting that a non-negligible percentage of the faults in the MIPS-like processor (2 429 of 268 424 according to TetraMAX in the present work) are undetectable, and thus never produce any misbehavior. A high percentage of untestable faults (3 291 of 111 398) has also been reported by [17]. The table presents both the Fault Coverage including all the faults (fc = detected faults / all faults) and the Test Coverage that excludes the undetectable faults (tc = detected faults / detectable faults).

The TetraMAX results presented in Table 6.3 also include some Possibly Detected (PT) faults. This kind of faults usually correspond to portions of the circuit that for any reason produce unknown (X) values in the simulation results. Such a fault will be activated or not, depending on the actual value taken by the node during test execution. As this actual value is unknown at simulation time, the fault is labeled as Possibly Detected. To consider the worst case, all the faults labeled as PT will be considered as Not Detected when computing coverage or in other words, using the TetraMAX terminology, the *partial credit* (percentage of Possibly Detected faults assumed as Detected) will be set to 0.

The results in Table 6.3 show a very good performance of the Memory Content observation method S4. Only about 3% of coverage reduction is produced when changing from S3 (90.40 %) to S4 (87.18 %), showing that the test program was able not only to propagate the fault effects up to the system bus, but also in most of the cases to preserve this effect as a difference in the final memory content. To compensate this coverage reduction, performance counter and debug interface observation methods can be used.

Table 6.4 shows the coverage obtained when combining the memory content observation solution (S4) with each of the solutions proposed here (S5, S5* and S6). The most interesting result is that the addition of these solutions allow a significant increase in the achievable Fault Coverage.

Table 6.4: Combining S4, S5, S5* and S6

	S4	**S4+S5**	**S4+S5***	**S4+S6**	**S4+S5+S5*+S6**
DT	231 895	238 018	238 046	240 384	240 488
incr.	-	6 123	6 151	8 489	8 593
tc	87.18 %	89.48 %	89.49 %	90.37 %	90.41 %
fc	86.39 %	88.67 %	88.68 %	89.55 %	89.59 %

For example, a significant amount (8 489) of the faults detected by S6 is possibly detected (880) or not detected (7 609) by S4. Hence, the total number of faults combining the two mechanisms (reported in the S4+S6 column) is higher than both of them, getting very close to what we can observe with processor-level observation (S3), which is not usable in an in-field test scenario. This result can be explained by recalling that the dbgm_branch mechanism allows accessing information about the internal behavior of the processor.

Similar results are obtained using any of the two performance counter based solutions S5 and S5*. The observation solution based on counting incorrect branch predictions (S5) provides 6 123 faults not detected by S4, most of them (5 061) from inside the Branch Prediction Unit. The timer based observation solution (S5*) adds 6 151 not detected by S4.

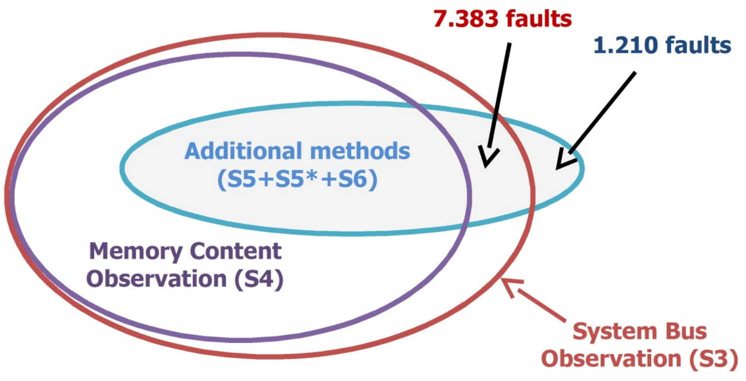

7.383 faults
1.210 faults
Additional methods (S5+S5*+S6)
Memory Content Observation (S4)
System Bus Observation (S3)

Figure 6.7: Full MIPS-like processor test case: sets of faults detected by the different observation methods.

The rightmost column in Table 6.4 presents the obtained coverage when combining all the observation methods (S4, S5, S5* and S6). Note that the results (240 488 faults detected, 90.41 % test coverage) are even better than the obtained with System Bus observation (240 471 faults detected, 90.40 % test coverage). This is explained because the proposed observation methods provide information from internal signals, not available at the processor outputs. Figure 6.7 shows the relation between the sets of faults detected by each method. The proposed methods add a total of 8 593 faults not detected by Memory Content observation (S4), 1 210 of which also escape to the System Bus observation method (S3).

Table 6.5: MIPS-like processor internal modules description

	Faults	[%]	**Description**
u1_pf	2 244	0.84	Address calculation stage
u2_ei	1 876	0.70	Instruction extraction stage
u3_di	8 468	3.15	Instruction decoding stage
u4_ex	**165 876**	**61.80**	Execution stage
u5_mem	3 394	1.26	Memory access stage
u6_renvoi	3 964	1.48	Bypass unit
u7_banc	**45 274**	**16.87**	Register bank
u8_syscop	8 524	3.18	System coprocessor
u9_bus_ctrl	1 422	0.53	Bus controller
u10_predict	**27 354**	**10.19**	Branch prediction
interconnect	28	0.01	Connections between modules
whole_circuit	268 424	100.00	

In order to better assess each observation method strengths and weaknesses, the detection and coverage data was obtained for each internal module of the

MIPS-like processor. Table 6.5 lists the internal modules along with the amount of possible stuck-at faults in the module. The execution stage is by far the biggest module with about 60 % of the faults, followed by the register bank (a 32x32 register bank with two read and one write ports) with about 17 % and the Branch prediction unit with 10 % of them.

In Table 6.6 the column labeled S2/S3 presents the fault coverage (detected / total module faults) obtained for each module using the Processor level observation method, i.e. the one for which the test program was originally developed. This column shows that the test program best performances are obtained for the Execution unit (97.11 %) and the Register bank (96.27 %). These are the bigger modules and consequently have the bigger impact on the obtained overall coverage. On the other hand, the poorer performances are for the Address calculation (51.34 %) and Memory access (53.42 %) stages and for the Branch prediction unit (57.49 %), being the former the one that influences the most on overall coverage due to its greater size.

Table 6.6: Fault coverage of the different internal modules

		S2/S3	**S4**	**S5**	**S5***	**S6**
	Faults	fc [%]	coverage relative to S3 [%]			
u1_pf	2 244	*51.34*	*36.8*	70.2	70.1	**99.0**
u2_ei	1 876	72.87	88.3	85.8	85.5	96.1
u3_di	8 468	71.76	90.5	81.1	79.5	95.1
u4_ex	165 876	**97.11**	**99.7**	6.0	6.1	6.5
u5_mem	3 394	*53.42*	95.7	65.7	66.0	65.7
u6_renvoi	3 964	79.06	94.5	72.1	69.6	71.9
u7_banc	45 274	**96.27**	**99.9**	26.2	25.1	26.2
u8_syscop	8 524	63.96	**100.0**	1.0	1.0	1.0
u9_bus_ctrl	1 422	75.67	*72.7*	41.8	43.0	44.1
u10_predict	27 354	*57.49*	*61.3*	**93.4**	**93.4**	**95.8**
interconnect	28	25.00	**100.0**	**100.0**	**100.0**	**185.7**
whole_circuit	268 424	89.59	96.4	19.4	19.2	20.4

The next column labeled S4 presents the percentage of the faults detected by S3 that are also detected by S4. Note that as stated in section 6.3 *"Analysis of the Results of Test Cases #1 and #2"*, all the faults detected by S4 are also detected by S3. This column shows that in the modules that are well covered by S3, like the Execution stage and the Register bank, the Memory Content observation method S4 covers almost all the faults covered by S3. It also covers the same faults covered by S3 in the Coprocessor system and the connections between modules. However, it has a poor performance in modules like the Address calculation stage (36.8 % of the faults covered by S3), the Branch Prediction unit (61.3 %) and the Bus controller unit (72.7 %).

The rightmost three columns in Table 6.6 present the coverage obtained by

Table 6.7: Fault coverage increment when adding other observation methods to Memory content observation

	S5	**S5***	**S6**	**all**	**S5**	**S5***	**S6**	**all**
	[# of faults]				[% of module faults]			
u1_pf	490	489	**716**	**716**	**21.8**	**21.8**	**31.9**	**31.9**
u2_ei	5	5	113	113	0.3	0.3	6.0	6.0
u3_di	255	250	**1 113**	**1 117**	3.0	3.0	**13.1**	**13.2**
u4_ex	126	176	**860**	**936**	0.1	0.1	0.5	0.6
u5_mem	15	19	16	20	0.4	0.6	0.5	0.6
u6_renvoi	165	139	159	165	4.2	3.5	4.0	4.2
u7_banc	0	0	39	39	0.0	0.0	0.1	0.1
u8_syscop	0	0	0	0	0.0	0.0	0.0	0.0
u9_bus_ctrl	6	16	30	40	0.4	1.1	2.1	2.8
u10_predict	**5 061**	**5 057**	**5 437**	**5 441**	**18.5**	**18.5**	**19.9**	**19.9**
interconnect	0	0	6	6	0.0	0.0	21.4	21.4
whole_circuit	6 123	6 151	8 489	8 593	2.3	2.3	3.2	3.2

S5, S5* and S6, also as a percentage of the faults detected by S3. But unlike the case of S4, observation methods S5, S5* and S6 include signals not observed by S3 and hence they allow the detection of faults uncovered by S3. As a consequence the figures in these three columns can be greater than 100 % as is the case in interconnect row using observation method S6. The results show that these new methods are complementary with S4 as they perform better than S4 in modules like the Address calculation stage and the Branch prediction unit where S4 coverage is bad.

Finally, Table 6.7 shows the coverage increment obtained by adding each of S5, S5* and S6 to the Memory content observation method S4, detailed by module. The coverage increments are presented both as the amount of faults and as a percentage of the module faults. The rightmost column shows the coverage increment when adding all the three methods. The main contributions to the coverage enhancement come from the Branch Prediction unit, and from the Decode instruction, Execution and Address calculation stages. A slightly better performance is observed in S6 if compared with S5 and S5*, produced mainly by the better performance in the instruction decoding and execution stages.

6.5 Some Lessons Learned

6.5.1 Setting the fault simulator observation times

The fault simulator used to assess the fault coverage in all of the three Test Cases was TetraMAX by Synopsys. To run the fault simulation experiment the simulator must be provided with a model of the system, the waveforms to be applied to the primary inputs of the system and the list of the faults that are to be simulated. It is also necessary to indicate which of the primary outputs of the system must be observed to detect a difference between the normal and faulty behavior of the system, and when must these indicated outputs be observed.

Regarding the last point (i.e. when to observe the outputs) the used simulator accepts two options: either setting periodic observation, hence allowing to emulate the observation of the outputs at each clock, or setting the observation synchronized with the rising or falling edge of another signal. Configuration is done through the *set_patterns* command, using the parameter *-strobe_period* to set periodic observation (for example *-strobe_period* { *200 ns* }) or any of the parameters *-strobe_falling* or *-strobe_rising* to set observation synchronized with a signal (for example *-strobe_falling write_pulse_signal*). In both cases an offset can be configured to delay the starting of the outputs observation using the parameter *-strobe_offset* (for example *-strobe_offset* {3000000 ns}). The offset parameter can be used to delay the outputs observation until the finalization of the test program execution.

During the early stages of the work on observation techniques, It was assumed that when using the *-strobe* variants of the command the calculation of the observation instants is performed independently during the simulation of each fault. In other words, It was assumed that for each fault the edge position of the strobe signal is determined according to the faulty behavior of the system affected by the fault being simulated.

However, what the fault simulator does is to set the observation instants according to the edge positions of the strobe signal in the fault free system, and consequently uses the same observation instants for all the faults.

Following the incorrect assumption described above, we attempted to use the strobe mechanism to observe the values written to memory by our system. This was used to obtain an estimation of the signature of the values written to memory for S4 observation method in Test Case #2, and in the observation method denoted as *mem* in the preliminary results of Test Case #3 published in [27].

For Test Cases #1 and #3 the problem was fixed later by including the memory into the system simulated by the fault simulator. In this way the final memory content can be observed as described in section 6.1 *"Test Case #1: Branch Prediction Unit"*.

In Test Case #2 this was not possible because of the bigger sizes of both the memory and the whole system. Consequently the coverage values corresponding to observation method S4 may be optimistic because in the presence of a fault affecting the timing the data buses are sampled in the instant that the transfer is done in the fault free system, instead of sampling it when the transfer is really done in the system affected by the fault.

6.5.2 Some comments on migrating a test program from end-of-manufacturing to in-field test scenario

The present section shows an example of the inconveniences that can happen when porting a test program conceived for end-of-manufacturing test to an in-field test scenario. The development of a test program requires an important effort from a test engineer. Consequently, if a test program was already developed, there is a strong pressure to adapt and reuse this existing program instead of developing a totally new one. This is true even if the new scenario is not exactly the same for which the test program was originally developed.

As already mentioned, this was the situation in the test program for Test Case #3. It was developed for an end of manufacturing scenario, assuming full observability over all the processor outputs. When adapting it to be used in a memory content observation solution, the test program was examined to identify all the instructions that write information to memory. A total of about 50 memory store instructions were identified. The target address in each one of these instructions was modified to assure that it writes its data in a different address in RAM, thus avoiding the attempt to write data to ROM or unused parts of the memory space, and avoiding also overwriting the values written by another part of the program. The coverage results of this initial version of the modified test program are shown in Table 6.8

Table 6.8: Full MIPS-like processor test case: initial version of test program fault simulation results

	S2/S3	**S4**	**S5**	**S5***	**S6**
DT	**240 111**	**107 229**	47 475	47 019	49 232
PT	988	1 631	318	351	208
ND	24 896	157 135	218 202	218 625	216 555
UD	2 429	2 429	2 429	2 429	2 429
total	268 424	268 424	268 424	268 424	268 424
tc	**90.27**	**40.31**	17.85	17.68	18.51
fc	89.45	39.95	17.69	17.52	18.34

In contrast with Test Case #1, which aims to cover only the faults inside a performance related module like the branch prediction unit, Test Case #3 targets the whole processor. In those conditions, a coverage reduction so large, from about 90% to about 40% when moving from a processor-level (S3) to a memory content observation solution (S4), was an unexpected result. A certain coverage reduction is expected because performance faults are poorly covered by S4, but a reduction of 50% would imply that about half of the faults are performance faults, i.e., they affect only the performance and not the calculation results.

The main reason for the coverage reduction was a misunderstanding of the test program behavior. Initially the fact that each store instruction can be executed several times inside a loop was not considered. An execution trace showed that a total of about 3500 store operations are performed during the whole test program execution. But only the last write operation into each of the 50 different memory addresses was affecting the final memory content, while the previous store operations on the same address produce no effect.

In order to enhance S4 results the test program was modified again having in mind that the intermediate results should modify the final memory content. Two alternative modifications that keep the redesign effort in a moderate level were evaluated preliminarily. One consists in using an internal register as a memory pointer and incrementing it to assure that each new store operation targets a different memory location, consequently requiring a larger RAM memory. The other alternative consists in substituting each store instruction with a load-modify-write sequence in order to produce a signature of the sequence of data written to each address, but at the cost of a strong impact on test duration. The results presented in the previous section correspond to the first of these two solutions that was the one finally implemented, obtaining a much better test coverage of about 87% (instead of about 40%) as seen in the previous section.

This example illustrates the kind of errors that can be made when migrating to an in-field test scenario a test program originally developed to be applied at end of manufacturing. It also shows the strong impact that these errors can produce in the fault coverage obtained by the migrated test program.

Part IV

Conclusions

Chapter 7

Conclusions

7.1 Summary

The initial part of the present thesis explores fault tolerance techniques to protect a computing system implemented on an FPGA against SEUs. The goal here is to harden computing intensive applications executed by a commercial-off-the-shelf processor implemented on SRAM-based FPGAs, as a way to enable its use in safety-critical applications. Computing systems based in modern processor cores synthesized on an FPGA have a series of advantages in terms of performance, cost and flexibility if compared with radiation-hardened hardware. As a consequence there is a strong pressure to enable the use of this kind of processors at least in the less critical parts of systems that must operate in radiation environments.

An architecture is proposed based on replicating task execution two or more times, assuring memory segregation between the different tasks by the use of a memory protection unit. A watchdog is included to protect the system against hang conditions and ad-hoc hardware can be added to accelerate the comparison of the results of each replica.

A proof-of-concept implementation of the proposed architecture was developed using the Altera NIOS-II as processor core, the Altera Memory Protection Unit IP core for memory segregation and a standard timer core for the watchdog. Being most of the cores already available and validated, the design effort is dramatically reduced.

Fault injection experiments were conducted to validate the proposed architecture. The results outline its fault tolerance effectiveness, and the overhead analysis shows that it is effective in reducing the resource occupation when compared to N-modular redundancy, at an affordable cost in terms of application execution time. The analysis also allows for a good understanding of the system behavior in the presence of faults, and of the effectiveness of some fault detection mechanisms provided by the processor used.

The main part of the present thesis deals with different ways to enhance the effectiveness of in-field software-based self-tests. The first result presented in this part consists in a method to detect stuck-at faults in the snooping protocol cache coherence logic of a multi-core system. A test program was developed in order to excite and expose any stuck-at fault both on the validity bit associated with each cache block and on the comparators used by the snooping logic. The test program runs concurrently in all the processor cores in a coordinated way. It is derived from the functional specifications of the circuitry under evaluation only, and can therefore be reused on any circuit implementing the same coherence protocol. In order to practically validate the method and to better quantify its cost in terms of memory occupation and execution time, some experimental results were gathered using a multi-core system integrating a variable number of LEON3 cores. The results show its cost in terms of execution time, which grows linearly (and slowly) with the number of cores.

When adopting SBST for in-field test of a processor-based system, as it is often required by standards and regulations for safety-critical applications, the achieved fault coverage is often affected in a strong manner by observability issues. More in detail, the fault coverage is affected by the limited ways that an in-field program test has to observe the results and thus identify a difference between the faulty and fault-free situations.

First, the different solutions that can be adopted to observe the results of a test were presented, analyzing the inconveniences and advantages of each method. The analyzed methods include some of the methods typically adopted for in-field and end of manufacturing test and a set of methods proposed to enhance the detection of performance faults.

Then, three test cases were developed to gather quantitative data about the faults that can be detected using the different observation mechanisms. Two of the test cases focus on the faults inside an internal module of the considered processor: a Branch Prediction Unit in a MIPS-like processor-based system, and a Data Cache Controller logic module in a dual-processor system based on the LEON3 processor. The other test case considers the faults in the whole MIPS-like processor of the first case. The test cases are on one hand simple enough to allow for an acceptable experiment duration (several days in the more complex case) and on the other hand complex enough to believe that the conclusions drawn are quite general.

Extensive fault coverage figures were obtained for the three test cases. Results show that, as expected, fault coverage decreases with reduced observability. In other words, when reusing the same test program originally developed for one observation mechanism in a different one, the fault coverage may significantly change.

More importantly, we experimentally demonstrated that a carefully devised combination of observation mechanisms, based on checking the memory content at the end of the test program execution and on the information coming from the Performance Counters or the Debug Interface existing in many processors, allows to achieve a fault coverage figure not far from the maximum one. More generally, a suitable set of Performance Counters or Debug information may allow for achieving nearly the same fault coverage achieved when continuously observing all the processor outputs, without significant extra costs for their detection.

The experimental results also show that, for a given test program, the proposed observation methods that use Performance Counters or the debug infrastructure provide a better coverage of the internal modules more likely to be affected by performance faults. Meanwhile, the final memory content examination method performs better in the other modules. This complementarity explains the good results mentioned above.

A detailed description of several solutions to implement suitable fault simulation campaigns able to gather the required experimental data is also provided.

Although all the experiments described in Part III *"Functional test"* refer to the single stuck-at fault model, the approach followed in this thesis and the main conclusions drawn also apply to other fault models, including the transition delay.

7.2 Main Contributions

In the present section a commented summary of the main contributions of the thesis is presented.

7.2.1 Experimental validation of a time-redundancy fault tolerance mechanism

An experimental validation of the feasibility of achieving a safe system by using a mix of already available IP cores, thus minimizing the development time, was presented in chapter 2. The Altera NIOS-II [36] processor was used as the processor core, and the Altera Memory Protection Unit IP core was used for memory segregation. Being all the cores already available, the design effort is limited to the integration with a custom watchdog timer and a DMA controller. By exploiting already existing cores a robust system can be obtained, which can be used with a number of different FPGAs supporting the same cores. As a result, a general architecture is obtained which is highly portable and reusable.

7.2.2 A method to detect faults in the cache coherence logic of a multi-core system

A method to detect faults in the cache coherence logic of a multi-core system using a software based self test approach is presented in chapter 4. The cache coherence logic presents difficulties when tested because it requires coordinated actions from the different processors in a multiprocessor, shared memory system.

The method proposed is based on a test program that runs concurrently and was validated on a LEON3 multicore system.

7.2.3 A survey of test observation methods

A survey of the different solutions that can be adopted to observe the results of functional tests and a discussion of the advantages and limitations of each of them is presented in chapter 5. The focus is set on the in-field test of microprocessor based systems. Usually in this scenario most of the observation methods used in production test are not available, namely the methods associated with the use of testers and DfT infrastructure.

The analyzed methods are: module-level, processor-level, system bus, memory content, performance counters, and debug interface. This list includes the memory content observation at end of test, a method traditionally adopted for in-field software-based self-test, and several methods targeted at obtaining a better coverage of the performance faults. A few solutions that are hardly available in-field are also analyzed to take them as a reference of what can be obtained with better observability.

7.2.4 A set of experimental test cases

A set of test cases to quantitatively evaluate the benefits and cost of each of the different observability solutions identified is presented in chapter 6. A comparison of the fault coverage obtained using two different observation methods was already presented in [71]. However, to the best of our knowledge, the present work and its associated papers [26] [28] are the first report of extensive experimental results to compare the fault coverage that can be achieved with the different observation methods, giving a much wider panorama of the advantages and disadvantages provided by the different solutions.

7.2.5 Use of a conventional fault simulator to assess the effectiveness of in-field SBST

A detailed description of the use of a conventional fault simulator to compute the fault coverage achieved by a software based self test is provided in chapter 6, when describing the system setup of each one of the presented Test Cases. For every observation solution considered, a fault simulation campaign was run in order to determine whether a fault produces a difference in the observed outputs if compared with the fault free situation. In some cases, the introduction of slight modifications to the system was necessary in order to mimic the observation mechanism under analysis in the simulation environment of the commercial fault simulator used.

7.2.6 Examples of coverage variation when changing observation environment

Examples were provided showing that the fault coverage obtained with a test program developed for one observation method may significantly change when reusing it with a different observation method. This is particularly relevant because, as the development of a test program requires an important effort from a test engineer, there is a strong pressure to adapt and reuse an existing program, instead of developing a totally new one. A common situation is trying to reuse in-field —and consequently with reduced observability— a test program originally developed for end of manufacturing test. As shown in all the test cases in chapter 6 a coverage reduction occurs because of the reduced observability. Also, as commented in subsection 6.5.2, when porting the test program to the new environment it is easy to make some mistakes that may dramatically affect fault coverage.

7.2.7 Performance fault oriented observation methods

Several observation methods were proposed that provide a good coverage of performance faults, a class of faults that is poorly covered by the observation method traditionally used for in-field test of microprocessor based systems. The observation of several performance counters and debugging information was proposed in chapter 6 as a way to catch the faults escaping the traditional memory content observation method and the effectiveness of the proposed methods was evaluated by using fault simulation experiments. Also, most of the examples showed that a proper combination of these complementary methods allow for achieving nearly the same fault coverage achieved when continuously observing all the processor outputs, an observation method commonly used for production test but usually not available in-field.

7.3 Future Work

A first point that seems attractive to analyze more in detail is related with the compromise between computational cost and accuracy of the results when trying to assess the effectiveness of functional tests using a fault simulator. Fault simulation is a computing intensive process. In order to simulate the system behavior in the presence of each fault and verify if a difference can be detected at the observed outputs, the fault simulator must be provided with a model of the system under analysis and the waveforms of the inputs applied to the system. Until recently, fault simulators had limited capabilities to support behavioral simulation models, and as a consequence very often the system must be modeled at the gate level.

In an attempt to reduce computation costs, there is a strong pressure to reduce the size of the fault simulated system. As a consequence, it is a common practice to perform the fault simulation including only a small part of the system in the circuit provided to the fault simulator. For example in section 6.2 *"Test Case #2: Data Cache Controller"*, when evaluating the module level observation solution (Solution S1), the top module in the fault simulation experiments was only the data cache controller module in one of the processors. The waveforms of the applied inputs and expected outputs were previously obtained by means of a fault free simulation of the whole system, including the two processors in the system and the program and data memories. When doing so, some feedback loops between the outputs and the inputs of the simulated system may be suppressed, and consequently the simulation of the faulty system may not accurately reproduce the behavior of the complete system.

A relevant subject is to determine in which conditions this kind of simplifications can be done without affecting the fault coverage assessment. Also, in the cases when the evaluated fault coverage is modified it would be useful to know if the effect of the simplification is an increase or a decrease of the value of the coverage estimation. During the work in the thesis valuable experience has been gained, but still some work must be done to formulate the ideas correctly. With some modifications, the system setup used in chapter 6 can be used to validate these ideas and to obtain examples of interest.

A second aspect to consider for future activities is related to the usability of the proposed observation methods in the presence of external conditions that cannot be controlled by the test program. For example, in all the examples we have assumed that all the processors and peripherals use the same clock or are perfectly synchronized, and that there are no external interrupts. Most of the proposed performance counters (timer, number of bus accesses, number of cache hots or misses) may be affected by these external factors. In some situations some of these factors may be controlled or their effects may be filtered out. It is of interest to give a closer view of how the different proposed observation methods are affected by these external factors, or how easily these external factors can be avoided or filtered out. The different sensitivity to these external factors between the different observation methods can be a determinant factor when choosing which one to use.

7.4 Publication List

The following list enumerates the papers directly related with the thesis work. In [27], Boyang Du presents a harness to exploit the debugging infrastructure provided by ARM processors to observe the results of in-field Software-Based Self-Test. My role in this paper was to mimic the same functionality on a miniMIPS based system and to perform fault simulation experiments to obtain quantitative fault coverage figures to validate the proposed approach. In all the other papers the main contributions are part of my work on the present thesis.

- J. Perez Acle, M. Sonza Reorda, and M. Violante, "Implementing a safe embedded computing system in SRAM-based FPGAs using IP cores: A case study based on the Altera NIOS-II soft processor," in 2011 IEEE Second Latin American Symposium on Circuits and Systems (LASCAS), 2011, pp. 1-5 [24].

- J. Pérez Acle, R. Cantoro, E. Sanchez, and M. Sonza Reorda, "On the Functional Test of the Cache Coherency Logic in Multi-core Systems," in 2015 IEEE VI Latin American Symposium on Circuits and Systems (LASCAS), 2015, pp. 2-5 [25].
- B. Du, E. Sanchez, M. Sonza Reorda, J. Perez Acle, and A. Tsertov, "FPGA-controlled PCBA power-on self-test using processor's debug features," in 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2016, pp. 1-6 [27].
- J. Perez Acle, R. Cantoro, A. T. Hailemichael, E. Sanchez, and M. Sonza Reorda, "Observability solutions for in-field functional test of processor-based systems," in 2015 XXX Conference on Design of Circuits and Integrated Systems (DCIS), 2015, p. 6 [26].
- J. Perez Acle, R. Cantoro, E. Sanchez, M. Sonza Reorda, and G. Squillero, "Observability solutions for in-field functional test of processor-based systems: a survey and quantitative test case evaluation," Microprocess. Microsyst., vol. 47, no. Part B, pp. 392-403, 2016 [28].
- J. Perez Acle, E. Sanchez, and M. Sonza Reorda, "About Performance Faults in Microprocessor Core in-field Testing," accepted in 10th IEEE Latin American Symposium on Circuits and Systems (LASCAS), 2019, p. 4.

A paper on closely related subjects presented results of the master thesis of Jorge Barboza, tutored by the author of the present thesis [91].

- J. Barboza, J. Basualdo, and J. Perez Acle, "Auxiliary IP blocks for early dependability analysis of small processor based systems," in 2016 17th Latin-American Test Symposium (LATS), 2016, pp. 21-26 [91].

Esta página ha sido intencionalmente dejada en blanco.

Bibliography

[1] International Electrotechnical Commission (IEC), "IEC 61508 : Functional safety of electrical/electronic/ programmable electronic safety-related systems," 2010.

[2] International Organization for Standardization, "26262: Road vehicles - Functional safety," 2011.

[3] RTCA SC-180, "Design assurance guidance for airborne electronic hardware," *DO-254*, 2000.

[4] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor Software-Based Self-Testing," *IEEE Des. Test Comput.*, vol. 27, pp. 4–19, may 2010.

[5] J. Shen and J. A. Abraham, "Synthesis of Native Mode Self-Test Programs," *J. Electron. Test.*, vol. 13, no. 2, pp. 137–148, 1998.

[6] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-Line Functionally Untestable Fault Identification in Embedded Processor Cores," in *Des. Autom. Test Eur. Conf. Exhib. (DATE), 2013*, (New Jersey), pp. 1462–1467, IEEE Conference Publications, 2013.

[7] "Microcontroller self-test libraries (Safety Libs)."

[8] "Guidelines for obtaining IEC 60335 Class B certification for any STM32 application. Application note AN3307," Tech. Rep. April, STMicroelectronics, 2013.

[9] "Cypress AN204377. FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library," tech. rep., Cypress, 2014.

[10] "Renesas. Software add-ons."

[11] "Microchip. 16-bit CPU Self-Test Library User ' s Guide," tech. rep., Microchip, 2012.

[12] "arm Developer. Functional safety."

[13] W. Perez, D. Ravotto, E. Sanchez, M. Sonza Reorda, and A. Tonda, "On the generation of functional test programs for the cache replacement logic," in *Proc. Asian Test Symp.*, pp. 418–423, 2009.

[14] E. Sanchez and M. Sonza Reorda, "On the Functional Test of Branch Prediction Units," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 23, pp. 1675–1688, sep 2015.

[15] N. Karimi, M. Maniatakos, C. Tirumurti, and Y. Makris, "On the impact of performance faults in modern microprocessors," in *J. Electron. Test. Theory Appl.*, vol. 29, pp. 351–366, 2013.

[16] M. Hatzimihail, M. Psarakis, D. Gizopoulos, and A. Paschalis, "A methodology for detecting performance faults in microprocessors via performance monitoring hardware," in *2007 IEEE Int. Test Conf.*, pp. 1–10, IEEE, 2007.

[17] A. Riefert, R. Cantoro, M. Sauer, M. Sonza Reorda, and B. Becker, "On the Automatic Generation of SBST Test Programs for In-Field Test," in *2015 Des. Autom. Test Eur. Conf. Exhib.*, (Grenoble), pp. 1186–1191, 2015.

[18] J. Pérez Acle, "Prototipado en FPGAs para inyección de fallas. Aplicación a sistemas distribuidos sobre bus CAN," Master's thesis, Facultad de Ingeniería, Universidad de la República, 2005.

[19] J. Perez Acle, M. Sonza Reorda, and M. Violante, "Dependability analysis of CAN networks: an emulation-based approach," *Proceedings. 18th IEEE Int. Symp. Defect Fault Toler. VLSI Syst.*, pp. 537–544, 2003.

[20] J. Perez Acle, M. Sonza Reorda, and M. Violante, "Accurate dependability analysis of CAN-based networked systems," in *16th Symp. Integr. Circuits Syst. Des. 2003. SBCCI 2003. Proceedings.*, pp. 337–342, IEEE Comput. Soc, 2003.

[21] F. Corno, J. Perez Acle, M. Ramasso, M. Sonza Reorda, and M. Violante, "Validation of the dependability of CAN-based networked systems," in *Proceedings. Ninth IEEE Int. High-Level Des. Valid. Test Work. (IEEE Cat. No.04EX940)*, pp. 161–164, IEEE, 2004.

[22] F. Corno, J. Pérez Acle, M. Sonza Reorda, and M. Violante, "A multi-level approach to the dependability analysis of networked systems based on the CAN protocol," in *SBCCI 2004. 17th Symp. Integr. Circuits Syst. Des. (IEEE Cat. No.04TH8784)*, pp. 71–75, 2004.

[23] J. Perez Acle, M. Sonza Reorda, and M. Violante, "Early, accurate dependability analysis of CAN-based networked systems," *IEEE Des. Test Comput.*, vol. 23, pp. 38–45, jan 2006.

[24] J. Perez Acle, M. Sonza Reorda, and M. Violante, "Implementing a safe embedded computing system in SRAM-based FPGAs using IP cores: A case

study based on the Altera NIOS-II soft processor," in *2011 IEEE Second Lat. Am. Symp. Circuits Syst.*, (Bogotá, Colombia), pp. 1–5, IEEE, feb 2011.

[25] J. Perez Acle, R. Cantoro, E. Sanchez, and M. Sonza Reorda, "On the functional test of the cache coherency logic in multi-core systems," in *2015 IEEE 6th Lat. Am. Symp. Circuits Syst.*, pp. 1–4, IEEE, feb 2015.

[26] J. Pérez Acle, R. Cantoro, A. T. Hailemichael, E. Sanchez, and M. Sonza Reorda, "Observability solutions for in-field functional test of processor-based systems," in *2015 XXX Conf. Des. Circuits Integr. Syst.*, (Estoril), p. 6, IEEE, 2015.

[27] B. Du, E. Sanchez, M. Sonza Reorda, J. Perez Acle, and A. Tsertov, "FPGA-controlled PCBA power-on self-test using processor's debug features," in *2016 IEEE 19th Int. Symp. Des. Diagnostics Electron. Circuits Syst.*, pp. 1–6, IEEE, apr 2016.

[28] J. Perez Acle, R. Cantoro, E. Sanchez, M. Sonza Reorda, and G. Squillero, "Observability solutions for in-field functional test of processor-based systems: a survey and quantitative test case evaluation," *Microprocess. Microsyst.*, vol. 47, no. Part B, pp. 392–403, 2016.

[29] J. Perez Acle, E. Sanchez, and M. Sonza Reorda, "About Performance Faults in Microprocessor Core in-field Testing," in *10th IEEE Lat. Am. Symp. Circuits Syst.* (IEEE, ed.), (Armenia, Quindío, Colombia), p. 4, IEEE, feb 2019.

[30] M. Ceschia, M. Violante, M. Sonza Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, and A. Candelori, "Identification and classification of single-event upsets in the configuration memory of sram-based fpgas," *IEEE Trans. Nucl. Sci.*, vol. 50, pp. 2088–2094, dec 2003.

[31] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, "FPGA partial reconfiguration via configuration scrubbing," in *2009 Int. Conf. F. Program. Log. Appl.*, pp. 99–104, IEEE, aug 2009.

[32] D. K. Pradhan, *Fault-tolerant computer system design.* Prentice-Hall, Inc., feb 1996.

[33] "Xilinx TMRTool."

[34] M. Sonza Reorda, M. Violante, C. Meinhardt, and R. Reis, "A low-cost SEE mitigation solution for soft-processors embedded in Systems on Pogrammable Chips," in *2009 Des. Autom. Test Eur. Conf. Exhib.*, pp. 352–357, IEEE, apr 2009.

[35] M. Pignol, "DMT and DT2: Two Fault-Tolerant Architectures developed by CNES for COTs-based Spacecraft Supercomputers," in *12th IEEE Int. On-Line Test. Symp.*, pp. 203–212, IEEE, 2006.

[36] "Altera Corporation - NIOS-II Processor Reference Manual," tech. rep., Altera Corporation, San Jose, CA, USA, 2009.

[37] N. Foutris, D. Gizopoulos, J. Kalamatianos, and V. Sridharan, "Measuring the performance impact of permanent faults in modern microprocessor architectures," in *2013 IEEE 19th Int. On-Line Test. Symp.*, pp. 181–184, IEEE, jul 2013.

[38] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 20, no. 3, pp. 369–380, 2001.

[39] J. Jian Shen and J. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," in *Proc. Int. Test Conf. 1998 (IEEE Cat. No.98CH36270)*, pp. 990–999, Int. Test Conference, 1998.

[40] S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors," *IEEE Trans. Comput.*, vol. C-29, pp. 429–441, jun 1980.

[41] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS - a microprocessor functional BIST method," in *Proceedings. Int. Test Conf.*, pp. 590–598, IEEE, 2002.

[42] L. Fournier, Y. Arbetman, and M. Levinger, "Functional verification methodology for microprocessors using the Genesys test-program generator. Application to the x86 microprocessors family," in *Proc. -Design, Autom. Test Eur. DATE*, pp. 434–441, 1999.

[43] S. Gurumurthy, M. Pratapgarhwala, C. Gilgan, and J. Rearick, "Comparing the effectiveness of cache-resident tests against cycleaccurate deterministic functional patterns," in *2014 Int. Test Conf.*, pp. 1–8, IEEE, oct 2014.

[44] G. Theodorou, S. Chatzopoulos, N. Kranitis, A. Paschalis, and D. Gizopoulos, "A Software-Based Self-Test methodology for on-line testing of data TLBs," in *Proc. - 2012 17th IEEE Eur. Test Symp. ETS 2012*, 2012.

[45] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-Based Self-Testing of Embedded Processors," *IEEE Trans. Comput.*, vol. 54, pp. 461–475, apr 2005.

[46] F. Corno, E. Sanchez, M. Sonza Reorda, and G. Squillero, "Automatic test program generation: a case study," *IEEE Des. Test Comput.*, vol. 21, no. 2, pp. 102–109, 2004.

[47] L. C. L. Chen and S. Dey, "Software-based diagnosis for processors," *Proc. 2002 Des. Autom. Conf. (IEEE Cat. No.02CH37324)*, 2002.

[48] D. Appello, P. Bernardi, M. Grosso, E. Sanchez, and M. Sonza Reorda, "Effective diagnostic pattern generation strategy for transition-delay faults in full-scan SOCs," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 11, pp. 1654–1659, 2009.

[49] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," in *Proc. - Des. Autom. Test Eur. Conf. Exhib.*, vol. 1, pp. 578–583, 2004.

[50] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. Sonza Reorda, "Test Program Generation for Communication Peripherals in Processor-Based SoC Devices," *IEEE Des. Test Comput.*, vol. 26, pp. 52–63, mar 2009.

[51] P. Bernardi, L. Ciganda, M. Sonza Reorda, and S. Hamdioui, "An Efficient Method for the Test of Embedded Memory Cores during the Operational Phase," in *2013 22nd Asian Test Symp.*, pp. 227–232, IEEE, nov 2013.

[52] M. Karpovsky and V. Yarmolik, "Transparent memory BIST," in *Proc. IEEE Int. Work. Mem. Technol. Des. Test*, pp. 106–111, IEEE Comput. Soc. Press, 1994.

[53] S. Di Carlo, P. Prinetto, and A. Savino, "Software-Based Self-Test of Set-Associative Cache Memories," *IEEE Trans. Comput.*, vol. 60, pp. 1030–1044, jul 2011.

[54] J. Sosnowski, "Improving Software Based Self - Testing for Cache Memories," in *2007 2nd Int. Des. Test Work.*, pp. 49–54, IEEE, dec 2007.

[55] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, "DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective-Testing in Multi-/Many-Core Microprocessors," *IEEE Trans. Comput.*, vol. 65, pp. 1453–1466, may 2016.

[56] J. Sosnowski, "Software-based self-testing of microprocessors," *J. Syst. Archit.*, vol. 52, no. 5, pp. 257–271, 2006.

[57] M. Riga, E. Sanchez, and M. Sonza Reorda, "On the functional test of L2 caches," in *2012 IEEE 18th Int. On-Line Test. Symp.*, pp. 84–90, IEEE, jun 2012.

[58] W. J. Perez H., J. V. Medina, D. Ravotto, E. Sanchez, and M. Sonza Reorda, "Software-Based Self-Test Strategy for Data Cache Memories Embedded in SoCs," in *2008 11th IEEE Work. Des. Diagnostics Electron. Circuits Syst.*, pp. 1–6, IEEE, apr 2008.

[59] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos, "Software-Based Self-Test for Small Caches in Microprocessors," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 33, pp. 1991–2004, dec 2014.

[60] B. O'Krafka, S. Mandyam, J. Kreulen, R. Raghavan, A. Saha, and N. Malik, "MPTG: a portable test generator for cache-coherent multiprocessors," in *Proc. Int. Phoenix Conf. Comput. Commun.*, pp. 38–44, IEEE, 1995.

[61] X. Quin and P. Mishra, "Automated generation of directed tests for transition coverage in cache coherence protocols," in *2012 Des. Autom. Test Eur. Conf. Exhib.*, pp. 3–8, IEEE, mar 2012.

[62] E. Sanchez and M. Sonza Reorda, "On the functional test of MESI controllers," in *2011 12th Lat. Am. Test Work.*, pp. 1–6, IEEE, mar 2011.

[63] M. L. Bushnell and V. D. Agrawal, *Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits.* Boston: Kluwer Academic Publishers,, 2000.

[64] CobhamGaisler, "LEON3 Processor."

[65] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach.* Elsevier Inc., 5th ed., 2012.

[66] H. Grigoryan, G. Harutyunyan, S. Shoukourian, V. Vardanian, and Y. Zorian, "Generic BIST architecture for testing of content addressable memories," in *2011 IEEE 17th Int. On-Line Test. Symp.*, pp. 86–91, IEEE, jul 2011.

[67] T.-Y. Hsieh, M. A. Breuer, M. Annavaram, S. K. Gupta, and K.-J. Lee, "Tolerance of performance degrading faults for effective yield improvement," in *2009 Int. Test Conf.*, pp. 1–10, IEEE, nov 2009.

[68] Intel Corporation, "Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes," 2011.

[69] M. Grosso, M. Sonza Reorda, M. Portela-Garcia, M. Garcia-Valderas, C. Lopez-Ongil, and L. Entrena, "An on-line fault detection technique based on embedded debug features," in *2010 IEEE 16th Int. On-Line Test. Symp.*, pp. 167–172, IEEE, jul 2010.

[70] W. Perez, J. Velasco, D. Ravotto, E. Sanchez, and M. Sonza Reorda, "A Hybrid Approach to the Test of Cache Memory Controllers Embedded in SoCs," in *2008 14th IEEE Int. On-Line Test. Symp.*, pp. 143–148, IEEE, jul 2008.

[71] T.-H. Lu, C.-H. Chen, and K.-J. Lee, "Effective Hybrid Test Program Development for Software-Based Self-Testing of Pipeline Processor Cores," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 19, pp. 516–520, mar 2011.

[72] FreescaleSemiconductor, "Freescale Semiconductor - e200z4 Power Architecture TM Core Reference Manual," tech. rep., Freescale Semiconductor, 2009.

[73] "AMBA TM AHB Trace Macrocell (HTM) Technical Reference Manual," tech. rep., ARM Limited, 2008.

[74] P. Bernardi, M. Grosso, M. Rebaudengo, and M. Reorda, “Exploiting an I-IP for both Test and Silicon Debug of Microprocessor Cores,” in *2005 Sixth Int. Work. Microprocess. Test Verif.*, pp. 55–62, IEEE, nov 2005.

[75] P. Bernardi, L. Ciganda, M. de Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda, and O. Ballan, “On-line software-based self-test of the Address Calculation Unit in RISC processors,” in *2012 17TH IEEE Eur. TEST Symp.*, pp. 1–6, IEEE, may 2012.

[76] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos, “Software-Based Self Test Methodology for On-Line Testing of L1 Caches in Multithreaded Multicore Architectures,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 21, pp. 786–790, apr 2013.

[77] W. Lindsay, E. Sanchez, M. Sonza Reorda, and G. Squillero, “Automatic test programs generation driven by internal performance counters,” in *Fifth Int. Work. Microprocess. Test Verif.*, pp. 8–13, IEEE Comput. Soc, 2004.

[78] H. O’Keeffe, “IEEE-ISTO 5001 TM-1999, The Nexus 5001 Forum TM Standard providing the Gateway to the Embedded Systems of the Future,” Tech. Rep. January, Ashling Microsystems Ltd., 2000.

[79] N. Stollon, *On-Chip Instrumentation Design and Debug for Systems on Chip*, vol. 53. Springer, 2013.

[80] “ARM CoreSight Debug and Trace.”

[81] A. Jutman, S. Devadze, I. Aleksejev, and T. Wenzel, “Embedded synthetic instruments for Board-Level testing,” in *2012 17TH IEEE Eur. TEST Symp.*, pp. 1–1, IEEE, may 2012.

[82] I. Aleksejev, A. Jutman, S. Devadze, S. Odintsov, and T. Wenzel, “FPGA-based synthetic instrumentation for board test,” in *2012 IEEE Int. Test Conf.*, pp. 1–10, IEEE, nov 2012.

[83] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso, and L. Entrena, “Online Test of Control Flow Errors: A New Debug Interface-Based Approach,” *IEEE Trans. Comput.*, vol. 65, pp. 1846–1855, jun 2016.

[84] ARM Corporation, “ARM CoreSight Program Flow Trace (PFTv1.0 and PFTv1.1) - Architecture Specification.”

[85] “CoreSight Technical Introduction A quickstart for designers,” Tech. Rep. August, ARM Limited, 2013.

[86] “miniMIPS.”

[87] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc, “GRLIB IP core user’s manual. Version 1.3.7 - B4144,” tech. rep., Gaisler research, 2014.

[88] R. Goldman, K. Bartleson, T. Wood, K. Kranen, C. Cao, V. Melikyan, and G. Markosyan, "Synopsys' open educational design kit: Capabilities, deployment and future," in *2009 IEEE Int. Conf. Microelectron. Syst. Educ.*, pp. 20–24, IEEE, jul 2009.

[89] "ARM CoreSight Architecture Specification." 2013.

[90] Xilinx, "Zynq-7000 All Programmable SoC Technical Reference Manual," tech. rep., Xilinx, 2015.

[91] J. Barboza, J. Basualdo, and J. Pérez Acle, "Auxiliary IP blocks for early dependability analysis of small processor based systems," in *2016 17th Latin-American Test Symp.* (IEEE, ed.), (Foz do Iguacu), pp. 21–26, IEEE, apr 2016.

Glossary

DUT Device Under Test

SBST Software-Based Self-Test

DfT Design for Testability

SoC System-on-Chip

ATE automatic test equipment

STL Self-Test Library

IP Intelectual Property

BPU Branch Prediction Unit

MIPS Microprocessor without Interlocked Pipeline Stages

FPGA Field Programmable Gate Array

ASIC Application Specific Integrated Circuit

SRAM Static Random Access Memory

SEU Single Event Upset

SEFI Single Event Functional Interruption

TMR Triple Modular Redundancy

DMT Duplex Multiplexed in Time

MPU Memory Protection Unit

EDAC Error Detection and Correction

RTL Register Transfer Level

ISA Instruction Set Architecture

BIST Built-In Self-Test

MESI Modified-Exclusive-Shared-Invalid (MESI), a cache coherence protocol

CCL Cache Coherence Logic

VB Validity Bit

MISR multiple-input signature register

ARM ARM, previously Advanced RISC Machine, originally Acorn RISC Machine

SPARC Scalable Processor Architecture SPARC

PeC Performance Counter

MMU Memory Management Units

TLB Translation Lookaside Buffer

DAP Debug Access Port

DP Debug Port

AP Access Port

APB ARM Peripheral Bus

AHB Advanced High-performance Bus

AXI Advanced eXtensible Interface

ATB AMBA trace bus

PTM Program Trace Macrocell

ETM Embedded Trace Macrocell

ITM Instrumentation Trace Macrocell

STM System Trace Macrocell

HTM AHB Trace Macrocell

FTM Fabric Trace Monitor

ETB Embedded Trace Buffer

TPIU Trace Port Interface Unit

VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

BTB Branch Target Buffer

VCD Value Change Dump

AMBA	Advanced Microcontroller Bus Architecture
SMP	Symmetric Multi-Processor
GPIO	General Purpose Input Output
UD	Undetectable
ND	Not Detected
DT	Detected
PT	Possibly Detected
fc	Fault Coverage
tc	Test Coverage
DMA	Direct Memory Access
e.g.	short for the Latin phrase exempli gratia, which means for example
i.e.	short for the Latin id est, which means that is, namely, or in other words

Esta página ha sido intencionalmente dejada en blanco.

List of Tables

Esta página ha sido intencionalmente dejada en blanco.

List of Figures