



*FP7-ICT-2013-11-619871*

# BASTION

*Board and SoC Test Instrumentation for Ageing and No Failure Found*

Instrument: Collaborative Project
Thematic Priority: Information and Communication Technologies

## Test optimization techniques (Deliverable D4.2)

Due date of deliverable: December 31, 2015
Ready for submission date: January 29, 2016

Start date of project: January 1, 2014 Duration: Three years

Organisation name of lead contractor for this deliverable: Politecnico di Torino

Revision 7.2

| Project co-funded by the European Commission within the Seventh Framework Programme (2014-2016) | | |
|---|---|---|
| Dissemination Level | | |
| PU | Public | ☒ |
| PP | Restricted to other programme participants (including the Commission Services) | ☐ |
| RE | Restricted to a group specified by the consortium (including the Commission Services) | ☐ |
| CO | Confidential, only for members of the consortium (including the Commission Services) | ☐ |

## Notices

For information, please contact Matteo Sonza Reorda, e-mail: matteo.sonzareorda@polito.it

This document is intended to fulfil the contractual obligations of the BASTION project concerning deliverable D4.2 described in contract 619871.

## Table of Revisions

| Version | Date | Description and reason | Author | Affected sections |
|---|---|---|---|---|
| 1.0 | November 5, 2015 | Structure created | M. Sonza Reorda | All |
| 1.1 | November 17, 2015 | Structure revised | M. Sonza Reorda | All |
| 2.0 | December 1, 2015 | Section 2.2 added<br>Section 4 added | M. Sonza Reorda<br>Piet Engelke | 2, 4, references |
| 3.0 | December 2, 2015 | Section 3 added<br>Section 4 improved | M. Sonza Reorda<br>Artur Jutman<br>Piet Engelke | 3, 4, references |
| 4.0 | December 11, 2015 | Section 2 completed | Jaan Raik | Section 2 |
| 5.0 | December 20, 2015 | Introduction and Conclusions added. General polishing | M. Sonza Reorda | Section 1, Section 5, all |
| 6.0 | January 17, 2016 | General polishing | E. Larsson, M. Sonza Reorda | All |
| 7.0 | January 21, 2016 | Section 2.3 improved | J. Raik | 2.3 |
| 7.1 | January 22, 2016 | Section 3.1 improved | S. Devadze, A. Jutman | Section 3.1 |
| 7.2 | January 29, 2016 | Preparing for submission | M. Sonza Reorda | All |

## Author, Beneficiary

S. Devadze, A. Jutman, A. Tsertov, Testonica Lab
M.-S. Beck, P. Engelke, J. Mejri, Infineon
J. Raik, TU Tallinn
B. Du, E. Sanchez, M. Sonza Reorda, Politecnico di Torino

## Executive Summary

This document reports on the activities performed within the Task T4.2 of the BASTION project, dealing with some advanced techniques for testing for faults at the board and the device level by combining embedded instruments and functional test. These techniques build on the results reported in the previous report (D4.1) produced by Task T4.1 and further extend and optimize them. The document first presents some techniques to reduce the size and duration of functional test, by dealing both with test programs intended to detected faults within a processor without reducing the achieved fault coverage, and with binary functional sequences for a generic sequential circuit. Secondly, we describe the test architecture for executing a mix of functional and structural test at the board level with the help of embedded instruments mapped on an FPGA. Finally, a solution to effectively support the functional test of an embedded analog-to-digital converter via some suitable hardware is described.

# List of Abbreviations

| | |
|---|---|
| ADC | Analog-to-Digital Converter |
| ATE | Automatic Test Equipment |
| BBIST | Board Built-In Self-Test |
| BIST | Built-In Self-Test |
| BSDL | Boundary-Scan Description Language |
| BS | Boundary-Scan |
| BST | Boundary-Scan Test |
| CAD | Computer Aided Design (also EDA) |
| CPU | Central Processing Unit, also Processor |
| DFT | Design For Testability |
| DNL | Differential Non-Linearity |
| DPM | Defects Per Million |
| DRAM | Dynamic RAM |
| DRC | Design Rule Check |
| EDA | Electronic Design Automation (also CAD) |
| EMS | Enhanced Manufacturing Services |
| FIFO | First In, First Out (data buffer) |
| FPGA | Field Programmable Gate Array |
| FP7 | European Union's 7th Framework Program |
| HW | Hardware |
| IC | Integrated Circuit |
| ICT | In-Circuit Test |
| IJTAG | Internal JTAG, a short name for IEEE 1687 standard and infrastructure collectively |
| INL | Integral Non-Linearity |
| IP | Intellectual Property (hardware module in FPGA or SoC) |
| JTAG | Joint Test Action Group; also Boundary Scan; often used as a short name of the IEEE 1149.1 standard and respective infrastructure including test access port and header on the board; |
| LSSD | Level-Sensitive Scan Design |
| NFF | No Fault Found or No Failure Found (also NTF) |
| NTF | No Trouble Found (also NFF) |
| PCB | Printed Circuit Board |
| PCBA | Printed Circuit Board Assembly |
| PDL | Procedural Description Language |
| POST | Power-On Self-Test |
| RAM | Random-Access Memory |

| | |
|---|---|
| RTD | Research and Technological Development |
| SAR-ADC | Successive-Approximation-Register ADC |
| SBST | Software-Based Self-Test |
| SIB | Segment Insertion Bit |
| SoC | System on Chip |
| SVF | Serial Vector Format |
| SW | Software |
| TDR | Test Data Register |
| UUT | Unit Under Test |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuit |

# Table of Contents

# 1 Introduction

A clear trend in industry, when dealing with testing of both devices and boards, is to complement structural test, mainly performed resorting to Design for Testability (DfT) techniques, with some sort of functional test, which directly applies stimuli at the functional inputs, and observes produced responses at the functional outputs [38]. The stimuli adopted for functional test are sometimes (when possible and feasible) specifically developed for this task, using some metric based on structural faults [3], and sometimes simply based on some application stimuli, mimicking the behavior of the device/system in the operation mode. The key reason for complementing structural test with functional one is that the former is often not able to catch some types of defects, which are likely to be detected by the latter.

Unfortunately, the above approach presents some serious drawbacks:

- When functional test is specifically developed (mainly at the device level) to guarantee a given fault coverage with respect to some fault model, the cost for test stimuli generation is typically huge, since very few solutions and tools exist to automate it
- When functional test is based on application stimuli (as often is the case for board test), it is often impossible to provide any quantitative evaluation of its fault/defect coverage capabilities
- The duration of functional test is often relevant, especially when compared with some types of structural test (e.g., based on BIST). This drawback is particularly critical when functional test is used for in-field test for guaranteeing the correct behavior of safety-critical applications.

In this deliverable, the above drawbacks are faced, and some solutions are presented to alleviate them. In particular, results are reported about the research performed by BASTION partners in the frame of Task 4.1 on test optimization techniques.

The described techniques build on those already presented in the previous report D4.2, and all aim at improving the defect coverage that can be achieved using functional approaches, possibly combined with some hardware support. The result is a contribution to decrease the impact of the so called *No Failure Found* (NFF) phenomenon (in which a product is labeled as faulty by the user, but passes all tests by the producer).

## *1.1 Structure of the document*

This report is structured as follows. First, Section 2 describes some compaction techniques (mainly suitable for device testing) able to reduce the size and duration of functional stimuli while preserving their fault coverage. Section 3 outlines some techniques to be used for board test in order to enhance the ability to detect faults at the Power-On Self-test (POST) by resorting to instruments mapped on possible FPGAs available on the board, as well as by using the debug features and interfaces existing in many processors. Section 4 deals with the test of Analog to Digital Converters, and presents a solution able to reduce the duration and increase the detection capabilities of a purely functional approach by complementing it with suitable hardware. Finally, Section 5 draws some conclusions.

# 2 Test Compaction

## 2.1 Introduction

When the first electronic circuits were developed, functional test was the only adopted solution. In the seventies, IBM started to adopt Level-Sensitive Scan Design (LSSD) to tame the growing difficulties in automatic generation of test stimuli [23]. In the following decades, scan became a popular solution, and the importance of functional test significantly shrank. Currently, Design DfT is widely adopted. However, functional test is still often used to complement other kinds of test, mainly because it allows to test the Unit Under Test (UUT) in exactly the same conditions existing during the operational mode. Areas of its adoption include end-of-manufacturing test of single devices, end-of-production test of Printed Circuit Board Assemblies (PCBAs), and in-field test of safety-critical systems [4].

In all these cases, a key parameter affecting the possibility of effectively adopting functional test is its length. Hence, several efforts have been made in order to compact functional test stimuli while preserving their ability to detect defects.

In this section we will report about the results achieved in this direction. In sub-section 2.2 a special case is considered, in which the UUT includes a processor. Hence, test stimuli take the form of a test program executed by the processor. In sub-section 2.3 the general case is considered, in which the UUT is a sequential circuit, and an existing functional test sequence is optimized, reducing its length.

## 2.2 Test Program Compaction

When processor-based systems are considered, functional test typically involves forcing the processor to execute a suitable test program, and then checking the processor behavior or the produced results. This solution, also called *Software-Based Self-Test* (or SBST) [3], has some important properties: first, it allows testing the system at-speed, since the test program can be executed at the same frequency adopted by application programs. Second, it can often be implemented without resorting to high-speed testers, since the test program is often uploaded in a memory accessible to the processor using low frequency interfaces, which can also be used to download the results from an internal memory. Moreover, since the test only consists of a piece of code, it can be easily adjusted to flexibly match different constraints: for example, the test can be adjusted to face new defects, or improved to provide diagnostic information [6]. Finally, it allows testing not only the single modules composing a system (processor, bus, memories, peripherals) but also their interconnections, as well as the system as a whole.

For the above reasons, functional test is widely adopted for the test of single devices (possibly corresponding to Systems on a Chip, or SoCs), boards and systems [4]; its adoption spans from end-of-manufacturing to incoming inspection and in-field testing. In the last case, the role of functional test is particularly important when considering safety-critical applications, for which standards and regulations may specify procedures to identify target fault coverages to be achieved in order to

guarantee a given level of safety. The constraints mandated by the ISO 26262 for automotive applications are a typical example.
On the other side, the main limitation of functional test for processor-based systems lies in the cost for the development of suitable test programs. Although some first solutions were proposed more than three decades ago [2], only in the last years research efforts have led to a comprehensive set of techniques allowing the test engineer to confidently develop test programs able to achieve good fault coverage figures. Recently, some approaches for automating the test program development task were also proposed [5], at least for small- and medium-sized processors.
Despite these recent efforts, the typical approach to test program generation is still mainly based on random or manual generation. In both cases, the results in terms of test program size and duration are far from optimal. When the test program has to be run in the field, it often exploits the time slots left idle by the application, whose duration is clearly limited. Hence, the duration of the test program execution is a very critical parameter, which strongly affects the applicability of the whole method. Similarly, when the test program is part of the end-of-manufacturing test process, its duration directly impacts the cost of the test process, and any reduction immediately turns into a money saving.
The use of dynamic test compaction during test program generation can reduce the size and duration of a test program. However, it will also increase the complexity of the test generation process. The solution considered in this sub-section is a static test compaction approach that does not interfere with the test generation process, and compacts the test program after it is generated. Among the static test compaction approaches, the simplest one is to omit parts of a test that are not necessary for achieving the fault coverage. In the context of test program compaction, this implies omitting instructions that are not necessary. This is the approach used here.
The effectiveness of the compaction process clearly depends on the method used to generate the initial test program: random or automatic generation often create test programs with limited efficiency, in which compaction may obtain good results; on the other side, a carefully crafted test program that was generated manually may often present little opportunity for compaction.
While the issue of compacting binary test sequences targeting a generic circuit has been deeply investigated (a summary will be given in sub-section 0), not so much work has been done concerning the compaction of test programs (instruction sequences running on processor-based systems).
Given a test program, logic simulation can be used to translate the program into a binary sequence. This binary sequence can then be compacted by one of the existing procedures. However, it may not be possible to translate the compacted sequence back into a test program (an instruction sequence). The goal of this work is to obtain a compact test program, without considering it as a binary sequence. Some of the challenges in compacting test programs, which do not exist for binary sequences, are the following.
First, the test programs work on systems including memories (which is rarely the case for binary sequences). Hence, compaction should take care of a more complex scenario when dealing with a test program. Moreover, test programs may include control flow instructions, which may force the execution of the same block of instructions more than once, or to skip a block. Additionally, the removal of an instruction may trigger special events (e.g., exceptions), which in some cases hang the whole system, or force it into hard-to-manage situations. Test sequence compaction does not have to care about this kind of situations.

As a conclusion, test program compaction tends to be a much more complex task than test sequence compaction.
There are few papers on test program compaction in the literature. In [7] a method was proposed, based on extracting from a test program an independent fragment (called a *spore*) whose iteration allows achieving a given fault coverage. The authors describe a method to identify the best sequence of spore activations that can achieve the same initial fault coverage, resorting to an evolutionary approach. The method is effective, but could only be applied under strict constraints (typically, the test of an arithmetic unit).
This work focuses on how to reduce the size of a general test program by omitting instructions from it. Although a reduction in size does not necessarily turn into a corresponding reduction in test duration, instruction removal tends to decrease test duration, too. The scenario we consider is the one (very common in practice) in which we do not have any specific information about the test program itself, and we aim at compacting it while preserving the initial fault coverage with respect to a given fault model.
We considered several possible solutions, starting from the brute-force one, corresponding to removing one instruction at a time from the test program, and checking whether the resulting fault coverage remains the same. This approach, although sometimes very effective, is typically highly CPU time consuming. For this reason, we propose smarter solutions based on instruction restoration: a group of instructions is initially removed from the test program; removed instructions are then restored one by one until the initial fault coverage is achieved. In this way we can significantly reduce the required computational effort, while still achieving significant compaction figures.
Experimental results are reported using a MIPS-like processor as a test case, and considering several test programs addressing faults in specific modules within the processor itself. Although our experiments were targeted to single stuck-at faults, the proposed approach is independent of the adopted fault model.

### 2.2.1 Previous work on test sequence compaction

The ability to compact a binary test sequence by omitting test vectors from it was first pointed out in [8]. It results from the fact that sequential test generation procedures cannot avoid including unnecessary test vectors in the sequences they generate, even if they use dynamic test compaction. Various procedures for the omission of test vectors from a binary test sequence were described in [8]-[16]. These procedures are reviewed next.
In general, the omission of the test vector at clock cycle $u$ of a test sequence $T$ can affect the detection of every fault that is detected by $T$ at clock cycle $u$ or later. To accept the omission of the test vector at clock cycle $u$, the procedure described in [8] performs fault simulation for these faults to ensure that they continue to be detected. To reduce the computational effort, the procedure described in [8] considers subsequences for omission. After finding that the test vector at clock cycle $u$ can be omitted, the procedure performs binary search to find the longest subsequence, starting at clock cycle $u$, that can be omitted without reducing the fault coverage.
A more efficient implementation of test vector omission is referred to as *restoration-based test vector omission* [10]. The procedure from [9] first omits all or most of the test vectors from the sequence. It then restores test vectors so as to restore the fault coverage. Whereas the decision to omit a vector requires simulation of all the faults that are affected by the omission, the decision to restore a test vector is made based on

simulation of a single fault. The restoration based procedure is, therefore, faster. By using parallel simulation, the restoration based test vector omission procedure described in [13] has the same computational effort as fault simulation of the sequence. In the procedure from [16], the initial omission of test vectors leaves a preselected fraction of the vectors in the sequence. The test vectors that are retained initially are selected randomly. Retaining a sufficient number of test vectors in the sequence initially slows down the convergence of the procedure, and allows it to reduce the sequence length over an increased number of iterations. As a result, shorter sequences can be obtained.
With the exception of [15], after the fault coverage is restored, the decision to omit a test vector from a sequence is final, and an omitted test vector will not be reintroduced into the compacted test sequence. To achieve higher levels of test compaction, the procedure described in [15] allows an omitted test vector to be reintroduced into the compacted sequence if this can be done without increasing the length of the sequence. In many cases, reintroducing an omitted test vector into the compacted test sequence allows other test vectors to be omitted such that the length of the sequence is reduced.
The procedure described in [14] modifies the test vectors in a sequence in order to improve the ability to omit test vectors from it. After it is not possible to omit additional test vectors from a sequence as it is, modifying the sequence can allow additional test vectors to be omitted. Modifying the sequence has a computational cost associated with it, since it needs to be done without reducing the fault coverage. A static test compaction procedure that modifies the sequence can be used when it is important to reduce the length of a functional test sequence below the length that can be achieved for the sequence as it is.
A higher level, assertion-based dynamic test compaction procedure was described in [17].

## 2.2.2 Proposed methods

### 2.2.2.1 Background

The problem we address in this work is described as follows. We assume that a test program TP is available for a given processor, composed of *n* instructions and starting from a well-specified state where all memory elements have a known value. TP achieves a fault coverage FC with respect to a given set of faults F composed of *f* faults. For the purpose of this work we consider single stuck-at faults, but the proposed solutions can be straightforwardly extended to other fault models as well. Similarly, in this work we assume that a fault is detected when it generates a difference compared with the fault-free system on any bus signal. Other detection mechanisms (e.g., looking at the memory content at the end of the test program execution) may be adopted, without impairing the effectiveness of the proposed techniques.
Our problem is to find a new test program TP', composed of a minimal subset of the instructions in TP, starting from the same initial state and achieving the same fault coverage FC. In addition, the order of the instructions in TP is not changed. For the purpose of this work we do not explicitly target the test program execution time; however, it is clear that it tends to decrease when reducing the number of instructions in TP.
Additional constraints may also exist on the behavior of the processor during the execution of TP (and TP'). In the rest of this sub-section we assume that the original

test program does not trigger any exception and does not enter any infinite loop, and we enforce the same conditions on the compacted one.

#### 2.2.2.2 Compaction by instruction removal (A0)

The most straightforward solution to test program compaction is based on instruction removal, following the idea proposed in [8] for binary test sequences: we denote this algorithm as A0.

Under A0, one instruction $I_i$ is selected and removed from TP. If an exception occurs, the resulting program is not correct (for example by removing a target label) or it enters an infinite loop, $I_i$ is restored. Otherwise, the resulting test program is fault simulated. If the achieved fault coverage is lower than the original one, $I_i$ is restored. Otherwise, $I_i$ is permanently removed. The process is then repeated with another instruction. The process is finished when the removal of any further instruction either causes an exception, or forces the program into an infinite loop, or reduces the achieved fault coverage, or when a maximum number of iterations has been performed.

We should note that the behavior of A0 is strongly dependent on the order used to select instructions: changing this order may change the results significantly. A random order is used in the experiments reported in this sub-section.

The computational cost of A0 depends on the number of iterations: at each iteration, at most *n-1* instructions are fault simulated with respect to the whole set of *f* faults. Clearly, the total effort can be reduced by only fault simulating at each iteration the instructions following the removed one (let us call it $I_i$) and the faults which are not yet detected, starting from the status at the end of the execution of instruction $I_{i-1}$. However, this requires

- saving the status of the good system and the one of the system in the presence of each of the still undetected faults at the end of the execution of $I_{i-1}$; and
- restoring the good and each of the faulty statuses, and then performing the fault simulation of the instructions in TP, starting from $I_{i+1}$.

If the considered system is complex, the cost for the two operations above may be significant. By storing only the detecting instruction of each fault, it is possible to simulate (under the complete test program) only the faults that are detected by an instruction $I_j$ such that $j \geq i$. A fault that is detected by an instruction $I_j$ such that $j < i$ does not need to be simulated.

#### 2.2.2.3 Restoration-based algorithm (A1)

A significant advantage in terms of CPU effort can be achieved by the following algorithm (denoted as A1), which is based on first removing a given block of instructions, and then restoring them one at a time until the original fault coverage is obtained.

This is similar to the restoration based procedure from [9], but with the difference that relatively small blocks of instructions are removed one at a time, instead of initially removing all or most of the instructions. In the case of a test program, removing all or most of the instructions may lead to exceptions during the restoration process, and preventing the exceptions may require the restoration of a large number of instructions. This issue does not exist with binary sequences. It is avoided with test programs by removing small blocks of instructions and restoring instructions from every block immediately after it is removed in order to restore the fault coverage.

A pseudo-code of the algorithm is shown in Fig. 1. In the proposed algorithm the original test program is split into *m* segments (step (2)). For every segment, we first remove all the instructions it is composed of, thus typically reducing the fault coverage. The set of faults which may not be detected any more is called $\Phi_i$. These are the faults that are detected by the segments $S_i$ to $S_{m-1}$. Then, we start restoring one instruction at a time until all the faults in $\Phi_i$ are detected again.
Algorithm A1 is also based on a number of iterations, each corresponding to a fault simulation experiment. However, the advantage of A1 over A0 lies in the fact that the number of iterations is lower than the number of instructions (like in A0), because iterations corresponding to instructions that do not need to be restored (because the fault coverage has already been restored) are not performed.

(1) Let F be the set of faults detected by TP
(2) Let us split TP into m segments $S_0$ to $S_{m-1}$
(3) For every segment $S_i$, starting from the last one
(3a) Identify the set of faults $\Phi_i$ detected by the instructions in the segments from $S_i$ to $S_{m-1}$
(3b) Let $TP_i$ be the test program obtained by removing from TP all instructions belonging to $S_i$. Assign TP = $TP_i$. If TP detects all the faults in $\Phi_i$, goto (3).
(3c) Select one instruction $I_i$ belonging to $S_i$
(3d) Fault simulate TP U $I_i$ wrt all the faults in $\Phi_i$. Assign TP = TP U $I_i$.
(3e) If all the faults in $\Phi_i$ are detected AND no exception is triggered AND the program does not enter an infinite loop
(3e.i) goto (3)
else
(3e.ii) goto (3c)

Fig. 1. Pseudo-code for algorithm A1

Segments are selected starting from the last one in the program. The advantage of this approach is that it requires only the simulation of faults belonging to the selected segment and the following ones, reducing the total CPU effort. Several versions of A1 are possible, depending on

- how the segments are defined (step 2); and
- how the instructions to be restored are selected (step 3c).

In the following, different policies for each point are considered.

## 2.2.2.4 Variants of A1

### *2.2.2.4.1 Segment selection*

Segments can be defined in several ways. We explored two basic solutions:

- Fixed-sized segments: step 2 simply partitions the original test program into equally sized segments, each composed on *k* instructions (apart from the last one, which may be shorter). The adoption of this solution leads to algorithms

having the suffix Fk (e.g., $A1_{F5}$), where *k* is the value selected for the segment size.

- Variable-sized segments: step 2 first performs fault simulation, and identifies for each fault the instruction which first detects it; we call this instruction the Detecting Instruction for fault $f_i$, or $DI_i$. Thus, a detecting instruction is one where the fault coverage is increased. Segments are now defined as the sequences of instructions in TP between one DI and the following one. The detecting instructions are assumed to be necessary to achieve the final fault coverage, and hence the algorithm does not try to remove them. The adoption of this solution leads to algorithms having the suffix V (e.g., $A1_V$).

#### *2.2.2.4.2 Instruction selection*

We explored three possible solutions for the order followed to restore instructions within a segment:

1. Random: at every iteration, we select a random instruction and evaluate whether its restoration allows us to obtain the target fault coverage. The adoption of this solution leads to algorithms having the suffix R (e.g., $A1^R$).
2. From the beginning of the segment: instruction restoration happens starting from the beginning of the segment. Hence, the first instruction is first restored, then the second, and so on. The adoption of this solution leads to algorithms having the suffix B (e.g., $A1^B$).
3. From the end of the segment: instruction restoration happens starting from the end of the segment. Hence, the last instruction is first restored, then the last but one, and so on. The adoption of this solution leads to algorithms having the suffix E (e.g., $A1^E$).

#### *2.2.2.4.3 Iterating the compaction algorithm*

Each of the previously described algorithms can be iterated. This means that we can apply it a first time on the original test program and the run it again on the resulting compacted test program. In principle, this procedure can be iterated several times, until no more compaction can be achieved. We will see in the experimental results that it is preferred to apply a different version of the same algorithm at each iteration.

## 2.2.3 Experimental Results

### 2.2.3.1 Experimental setup

In order to experimentally validate the proposed techniques we implemented them in a prototypical tool written in Java (composed of about 1,200 lines of code). The tool interacts with Synopsys TetraMAX for single stuck-at fault simulation.

For our experiments we used a MIPS-like processor [18]: its architecture is based on 32-bit registers and addresses, and includes a five-stage pipeline, accounting for about 45k equivalent gates when synthesized (without the multiplier) using the FreePDK45 Generic Open Cell Library from NanGate [19]. We focused on the test programs developed to detect faults related to two units:

- The *Register Forwarding and Pipeline Interlocking unit* (RFPIU): this unit, which is contained in the execution stage, deals with data hazards, creating special paths among pipeline registers when required, thus reducing the number of pipeline stalls. The unit has 3,738 single stuck-at faults. Despite its relatively small size, testing this

unit is particularly complex, since it requires very specific sequences of instructions, which are able to activate the different types of data hazards [20].

- The *Decode unit* (DU): this unit interprets the codes of the fetched instructions and generates the control signals triggering the required operations on the processor data path. This unit has 7,502 stuck-at faults. The test of this unit is complex because it requires not only to execute the different instructions with the different instruction modes, but also to consider the behavior of the unit when illegal instructions are fetched.

For each of the two units we considered two test programs generated with different approaches:

- A program denoted P1 which has been manually generated resorting to the techniques described in [20] and [21], respectively.
- A program denoted P2 which has been automatically generated resorting to the μGP tool [22].

Details about the treated test programs are reported in Table I.

TABLE I. ORIGINAL TEST PROGRAM CHARACTERISTICS

| | Size [#instr] | FC % |
|---|---|---|
| RFPIU – P1 | 341 | 92,63 |
| RFPIU – P2 | 460 | 86,54 |
| DU-P1 | 68 | 71,80 |
| DU-P2 | 448 | 79,38 |

### 2.2.3.2 Results

We first evaluated the results that can be achieved using the algorithm A0. Instructions to be removed are randomly selected without repetition in order to avoid a second attempt to remove the same instruction. Since results are not deterministic, experiments were performed five times, and we took the average results, which are reported in Table II. The CPU effort is reported as the ratio between the run time required for test compaction and the run time required to fault simulate the original test program (which is in the order of a few seconds for the test cases we considered).

TABLE II. COMPACTION RESULTS FOR A0

| | Size [#instr] | | Compaction % | Cost |
|---|---|---|---|---|
| | Original | Compacted | | |
| RFPIU – P1 | 341 | 274 | 19.65 | 341 |
| RFPIU – P2 | 460 | 415 | 9.78 | 460 |
| DU-P1 | 68 | 45 | 33.82 | 68 |
| DU-P2 | 448 | 145 | 67.63 | 448 |

It is clear that the compaction capabilities of any algorithm strongly depend on the test program to be compacted: for A0 with the considered test cases they range from less than 10% to more than 67%. The computational cost directly depends on the length of the original test program.
We then considered the version of our tool implementing A1 and ran experiments aimed at understanding the effects of the different choices listed in Section III.D.

We started with the algorithm using fixed-sized segments and analyzed the effects of changing the size of the segments. We report in Table III the experiments performed on the program P1 for DU with instructions restored from the beginning of the segment: the segment size ranged from 2 to 20 instructions. Similar results can be obtained with P2 as well as on the RFPIU and with the other strategies in terms of restoration order. The results of Table III show that

- the compaction figure is lower than for A0, but the computational effort is significantly lower than for A0, with more than 50% reduction; and
- with smaller segments we could reach the best compaction figures; however, the cost for compaction is minimum when the segment size is equal to 5 instructions. This behavior can be explained partly with the different compaction figure (the more instructions we remove, the lower the computational cost), and partly with the following observation. Each segment requires an initial fault simulation when all its instructions have been removed (smaller segments will require a higher number of these fault simulation runs): hence, there is a fixed cost for the management of each segment, and the minimum computational effort can be obtained with segments composed of about 5 instructions.

Therefore, for the rest of this sub-section we will use $k=5$ (being $k$ the number of instructions composing each segment) whenever the fixed segment strategy is adopted.

TABLE III. COMPACTION RESULTS FOR $A1_{F2}^{B}$ TO $A1_{F20}^{B}$ ON THE PROGRAM P1 FOR DU

| | Size [#instr] | | Compaction % | Cost |
|---|---|---|---|---|
| | Original | Compacted | | |
| $A1_{F2}^{B}$ | 68 | 52 | 24 | 26 |
| $A1_{F3}^{B}$ | 68 | 52 | 24 | 25 |
| $A1_{F5}^{B}$ | 68 | 55 | 19 | 24 |
| $A1_{F10}^{B}$ | 68 | 57 | 16 | 28 |
| $A1_{F15}^{B}$ | 68 | 61 | 10 | 32 |
| $A1_{F20}^{B}$ | 68 | 58 | 15 | 25 |

In Table IV we report the results of the experiments we performed to compare the effectiveness of the different restoration orders. They show that there is no significant difference between the strategies restoring the instructions from the beginning or the end of the segment. On the other side, random restoration seems to be slightly less effective.

TABLE IV. COMPACTION RESULTS FOR $A1_{F5}^{B}$, $A1F_5^{E}$, $A1_{F5}^{R}$ ON THE PROGRAMS P1 AND P2 FOR DU

| | | Size [#instr] | | Compaction % | Cost |
|---|---|---|---|---|---|
| | | Original | Compacted | | |
| P1 | $A1_{F5}^{B}$ | 68 | 55 | 19 | 24 |
| | $A1_{F5}^{E}$ | 68 | 55 | 19 | 24 |
| | $A1_{F5}^{R}$ | 68 | 58 | 15 | 24 |
| P2 | $A1_{F5}^{B}$ | 448 | 246 | 45 | 142 |
| | $A1_{F5}^{E}$ | 448 | 240 | 46 | 142 |
| | $A1_{F5}^{R}$ | 448 | 258 | 42 | 142 |

Finally, we performed experiments aimed at assessing the effectiveness of the strategy with variable-sized segments with respect to the one with fixed-sized segments. Results reported in Table V show that the fixed-sized options achieve higher compaction figures, but require longer computation times.

As mentioned in Section 2.2.2.4.3, the algorithms we proposed can be applied iteratively. Experiments (whose details are not reported here for lack of space) show that in all the considered test cases we can achieve higher compaction figures by running the compaction algorithm a second time on the result produced by the first application. Moreover, better results can be achieved by diversifying the algorithm executed in the second run with respect to the one of the first iteration. For example, improved results could be achieved by running the algorithm $A1_{F5}^{B}$, followed by $A1_{F5}^{E}$ (we denoted this version as $A1_{F5}^{B+E}$). Clearly, the computational cost increases with each iteration, but this gives our approach some flexibility, that the user can exploit to further improve the level of compaction, if he/she can afford the extra computational time. Table VI reports the results obtained by running the $A1_{F5}^{B+E}$ algorithm on all the four test cases we considered. This algorithm is shown to be faster than A0, while the achieved compaction figures are generally only a bit lower (with the exception of RFPIU – P2, where $A1_{F5}^{B+E}$ is both faster and more effective than A0).

TABLE V. COMPACTION RESULTS FOR $A1_{F5}^{B}$ , $A1_{V}^{E}$ , $A1_{V}^{B}$ ON P1 AND P2 FOR DU

| | | Size [#instr] | | Compaction % | Cost |
|---|---|---|---|---|---|
| | | Original | Compacted | | |
| P1 | $A1_{F5}^{B}$ | 68 | 55 | 19 | 24 |
| | $A1_{V}^{E}$ | 68 | 61 | 10 | 22 |
| | $A1_{V}^{B}$ | 68 | 56 | 18 | 22 |
| P2 | $A1_{F5}^{B}$ | 448 | 246 | 45 | 142 |
| | $A1_{V}^{E}$ | 448 | 382 | 15 | 103 |
| | $A1_{V}^{B}$ | 448 | 322 | 28 | 103 |

TABLE VI. COMPACTION RESULTS FOR $A1_{F5}^{B+E}$

| | Size [#instr] | | Compaction % | Cost |
|---|---|---|---|---|
| | Original | Compacted | | |
| RFPIU – P1 | 341 | 281 | 18 | 257 |
| RFPIU – P2 | 460 | 394 | 14 | 361 |
| DU – P1 | 68 | 51 | 25 | 49 |
| DU – P2 | 448 | 164 | 63 | 214 |

### 2.2.4 Conclusions

This sub-section proposed some algorithms for reducing the size of an existing test program without reducing the achieved fault coverage. Different solutions have been explored, based both on instruction removal and on instruction restoration, which is faster and often achieves similar or even better compaction figures. Although compaction results clearly depend on the initial test program, on the test cases considered our algorithms can achieve up to 63% compaction without exploiting any knowledge about the strategy adopted for the generation of the test program itself. The algorithm can be easily tuned to match different constraints, either by targeting the maximum compaction figures, or by achieving smaller compaction figures with reduced computational effort.

## *2.3 Binary Sequence Compaction*

In the previous subsection we dealt with a UUT including a processor, and faced the issue of compacting a sequence of test stimuli corresponding to a test program. In this sub-section we will consider the case in which the UUT is a generic sequential circuit, and hence the test sequence is a sequence of test vectors. We will consider static compaction (hence, working downstream the test pattern generation step) and propose a new approach based on a greedy optimization function. Results on a large number of sequential benchmark designs showed that the approach is capable of compacting large test sets within a short run-time, achieving close-to-optimal results.

### 2.3.1 Problem definition

Let the *test set* T consist of *test sequences* $t_i \in T$, i = 1, ... , n. Each sequence $t_i$ contains in turn $L_i$ *test vectors*. The *test length* of sequence $t_i$ is given by the number of test vectors $L_i$ . The total test length of test set T can be viewed as a sum

$$\sum_{i=1}^{n} L_i$$

The set of faults $f_j$, j = 1, ..., m detected by T is denoted by F.
In binary sequence compaction, our task is to find values for the $L_i$ such that the above sum would be minimal while the number of faults that the test set T detects would still be |F|.
Consider the test set example shown in Fig. 2 that consists of three test sequences $s_1$, $s_2$ and $s_3$, respectively. Sequence $s_1$ consists of four test vectors covering fault $f_2$ at the third vector and $f_1$ at the fourth vector. Sequence $s_2$ consists of three test vectors covering $f_1$ at the first vector and $f_3$ at the third vector. Finally, sequence $s_3$ consists of four test vectors covering $f_2$ at the first vector, $f_3$ at the second vector and $f_4$ at the fourth vector.


**Fig. 2 Binary Sequence Compaction Example**

The test set *T* consists of *m* faults and *n* test sequences can be viewed as a matrix

$$T = \begin{pmatrix} t_{s_1,f_1} & t_{s_1,f_2} & \dots & t_{s_1,f_m} \\ t_{s_2,f_1} & t_{s_2,f_2} & \dots & t_{s_2,f_m} \\ \dots & \dots & \dots & \dots \\ t_{s_n,f_1} & t_{s_n,f_2} & \dots & t_{s_n,f_m} \end{pmatrix}$$

where $t_{si,fj}$ is equal to *k* if sequence $s_i$ covers fault $f_j$ at the *k*-th vector and zero if sequence $s_i$ does not cover fault $f_j$.
If we select *k* vectors from sequence $s_i$ then all the faults $\{f_j : k \geq t_{si,fj} > 0\}$ are said to be covered by these vectors. In our algorithm we remove the columns corresponding to the covered faults from matrix *T*. In addition, we must subtract *k* from all the non-

zero elements $t_{si,fj}$ of the row corresponding to the sequence $s_i$. Our task is to cover all the faults (i.e. columns of matrix $T$) by selecting the minimal number of vectors. This task belongs to the class of NP-complete problems.

## 2.3.2 Greedy optimization method

In the following, three relationships are implemented to prune the search space for the compaction algorithm.

### 2.3.1.1 Selecting essential vectors

If fault $f_j$ is detected by the $k$-th vector of test sequence $s_i$ and is not detected by any other sequence belonging to the test set then $k$ first vectors of sequence $s_i$ are called essential. After selecting the essential vectors, we remove them from the test sequences. In addition, we remove the columns corresponding to faults covered by these vectors from matrix $T$. This simple pre-processing step allows to significantly reduce the search space for the static compaction algorithm.

### 2.3.1.2 Removing dominated faults

Column $f_a$ is said to *be dominated by column* $f_b$ and will be removed from matrix $T$ if

$$\forall_{i=1}^{m} t_{s_i,f_b} \neq 0 \Rightarrow t_{s_i,f_a} \neq 0,\ t_{s_i,f_b} \geq t_{s_i,f_a}.$$

The motivation for removing dominated faults is the following. Let us assume that a fault $f_b$ dominates fault $f_a$. The above relationship shows that no matter by which sequence selection we cover fault $f_b$ we will also cover fault $f_a$. Since in order to achieve full matrix coverage we will have to cover $f_b$ in any case, column $f_a$ represents redundant information for the optimization problem.

### 2.3.1.3 Removing dominating sequences

A row corresponding to sequence $s_b$ is said to be a *dominating sequence* of $s_a$ and will be removed from matrix $T$ if

$$\forall_{j=1}^{n} t_{s_b,f_j} \neq 0 \Rightarrow t_{s_a,f_j} \neq 0,\ t_{s_a,f_j} \leq t_{s_b,f_j}.$$

A dominating sequence can be removed because the sequence dominated by it covers all the same faults in a shorter or equal vector range.
The developed method applies the above described steps of selecting essential vectors, removing dominated faults and dominated columns as far as possible. If the fault matrix cannot be pruned by these steps any further, then it switches to a greedy algorithm. The greedy selection function implemented in is described in the following. Let us denote by *Minrange*($f_j$) the minimal number of vectors that has to be selected from any test sequence in order to detect a fault $f_j$. Let *Maxrange* be the maximum *Minrange*($f_j$) of all the faults.

The selection function selects *Maxrange* vectors from the corresponding test sequence. If there exist multiple maximal *Minrange*($f_j$) values then the algorithm prefers the sequences that detect more faults in *Maxrange* first vectors.
The algorithm described allows calculating the lower bounds for the static compaction task. The meaning of the lower bounds is that they show that it is not possible to compact the test set to contain fewer vectors than the bound. Moreover, in the cases where the result found by the algorithm equals the lower bound we have proved that this result is the global optimum.
In current approach, the lower bound is determined by our technique with the number of vectors selected during the compaction process up to the first greedy selection, including the vectors chosen by that selection. The first greedy selection can be included due to the fact that, obviously, it represents the minimal number of vectors that are necessary in order to cover a previously uncovered fault $f_j$. All the alternative combinations of selecting vectors for covering $f_j$ must always result in a greater or equal number of vectors.

### 2.3.3 Experimental results

Experiments on 40 sequential test sequence examples showed that the approach presented in this sub-section is able to compact the sequences in average by 42.97%, whereas the upper bound proven by the implications was 44.49%. Furthermore, in 75% of the experiments the tool was able to prove using the lower bounds that a globally optimal solution was reached. Experiments run on an IBM System x3500 M3 7380 computer showed no run time higher than 530 seconds for the s38584 test example with 38k faults, 271 sequences and 8065 tests.
The tool has been integrated into the TUT's test tool framework Turbo Tester. In the future it is planned to apply it for the minimization of test sequences represented in the binary form.

# 3 FPGA-assisted in-field PCBA Test

Power-On Self-Test (POST) plays an important role in many systems, since it may detect faults arising during the life time of the product, thus increasing its dependability. POST may use different solutions, which should match the constraints of the environment the system is deployed in.

Functional test [25], [5] represents a commonly adopted solution for POST. More in general, functional test is adopted in many scenarios, at the device, board and system levels [4]. In some of them it complements other test steps, performed resorting to DFT. For example, when considering PCBA test, it is common to see functional test as the last step, mainly targeting dynamic defects. The importance of this kind of defects is significantly increasing in the last years, especially since they are considered one of the major contributors for NFF [26][32], thus raising the interest for any solution able to improve the achievable defect coverage.

Another scenario where functional test plays a key role is in-field test of PCBAs [27]. In this context, functional test is particularly attractive due to its relatively low implementation cost (no equipment required), low intrusiveness (no or limited changes in the PCBA design and low impact on the application) and flexibility (since functional test is typically based on forcing the processor to execute a suitable test program, possibly mimicking some specific application code). A commonly adopted solution consists in launching at the power-on the execution of a functional test targeting all the critical modules in the system, or, alternatively, some specifically crafted test able to make observable the highest percentage of defects. Functional test is normally based on a sequence of instructions to be executed by the in-system processor(s), aimed at exciting possible defects in the processor itself or in any other device on the board, as well as in the interconnect. Therefore, this kind of test is sometimes referred to as *Software-Based Self-Test*, or SBST [3].

Major limitations to the effectiveness of functional test include

- The difficulty in assessing the achieved defect coverage: although many efforts have been done to introduce high-level metrics, their correlation with real defect coverage is still a matter of discussion and a hot research topic [33];
- The cost for creating suitable functional stimuli; a lot of work has been done to automate this part, or at least to provide guidelines for the test engineer in charge of developing suitable functional tests;
- The limited observability that we can obtain on the behavior of the system under test, both as a whole and in terms of its components.

The work described in this section specifically focuses on the last point, and proposes an approach based on combining the following several solutions. First of all, we propose to reuse for test purposes the features originally developed to support the debug of embedded software. In particular, these features currently provided by many processors often allow on-the-fly monitoring access during its normal operation (and without slowing it down) to service information on internal status and behavior. A typical example is tracing the sequence of instructions executed by the processor, writing them to ad hoc external interfaces.

Secondly, since the on-the-fly monitoring of the data flow produced by the debug interface can only be done externally, and hence by an ad hoc hardware, we propose to use a module (hereinafter also called Intellectual Property core, or simply IP) mapped on an FPGA device, which is assumed to be available on the board [28]. A key advantage of this approach is that the test is performed at the operational

frequency of the system, and is thus able to detect defects that can hardly be addressed with other solutions.
In this scenario, the FPGA plays the role of the embedded on-board test control and monitoring device. Since the FPGA can load one of multiple configurations stored in the flash memory device, the same FPGA can be used for supporting different test steps during the POST procedure, as well as for performing normal application-related functions during system's normal operation.
The POST scenario that we propose consists of three main phases:
1. FPGA self-test
2. FPGA-controlled board test
3. FPGA-monitored processor-based functional test.

FPGA self-test solutions are known from the literature [34][35], hence we leave this subject out of the scope of this work. FPGA-controlled board test is presented in detail in sub-section 3.1. The solution we propose is based on converting existing Boundary Scan manufacturing test (patterns, procedures, diagnostic routines) into the format that can be run by an FPGA. We demonstrate that such seemingly difficult conversion can be streamlined and that the dedicated instrument IP configuration including test patterns for a typical test occupies around 10% of FPGA resources in one of the smallest FPGA devices (our experiments were performed using Xilinx Spartan 3). Full automation of this approach is possible making it an effective and efficient in-field test solution. The related overhead is negligible, as the same existing FPGA is reused for test.
The processor-based functional test phase is introduced in sub-section 3.2. In order to assess the effectiveness and limitations of the proposed solution we first focused on a Zynq-7000 system by Xilinx, which integrates one or more ARM processors with a programmable module. We developed an IP core that can be mapped on the latter and is able to monitor the debug interface provided by ARM, as well as a small SW library to be integrated into the code of the functional test, so that the debug features are properly programmed at the beginning of the test, and then used to increase the observability during the test execution. In this way we could both demonstrate the feasibility of the proposed solution, and its (rather limited) cost in terms of required FPGA resources.
Since the lack of detailed information about the structure of the ARM processor clearly prevents us from computing the increase in defect coverage that can be achieved using our solution, we performed some fault simulation experiments on a MIPS-like processor for which the model is available: the same information produced by the ARM debug interface are extracted from this processor during the execution of the test, and the achieved fault coverage is computed. Results show the effectiveness of the proposed solution. Interestingly, they demonstrate that the stuck-at fault coverage that can be achieved is even higher than the one reachable using a corresponding test program in a scenario where all the processor outputs can be continuously monitored.
This section is organized as follows: Section 3.1 outlines the general board-level solution implementing the test and the IP supporting the reuse of Boundary Scan patterns developed for end-of-manufacturing test. Section 3.2 deals with the solution using the debug features: we first summarize the main debug features offered by ARM processors, with special emphasis on those we propose to use for test purposes. Then, we describe the IP we devised for monitoring the debug port and observing the behavior of the processor during test, and provide some figures about its implementation on a Zynq-7000 device. Finally, we report some figures derived from

fault simulation experiments performed on a MIPS-like processor, providing an evaluation of the increase in observability (and Fault Coverage) that can be obtained using the proposed technique. Section 3 draws some conclusions.

## 3.1 FPGA-controlled PCBA self-test

In this sub-section we will describe a technique for implementing *Board Built-In Self-Test* (BBIST) which is capable of carrying out in-field tests of PCBAs. The proposed method relies on the usage of FPGA devices that are frequently present on modern complex PCBAs. The idea behind BBIST is to embed Boundary Scan (BS) test procedures into an FPGA, so that BS test can be repeated each time an electronic device is powered-up. The method is intended to be fully autonomous and does not involve usage of any external measurement equipment.

The proposed technique does not require any extra FPGA device to be mounted on board, as the existing FPGA can be re-used. During normal (functional) mode of operation of an electronic device, this FPGA is loaded with functional design and performs its normal operational tasks. However, during the test stage the same FPGA is configured with a special IP which converts it into an embedded test controller. Despite certain amount of DFT still needs to be introduced, this overhead is extremely small in comparison with the overall complexity of an average board. Another advantage of BBIST is the capability to re-use Boundary Scan tests that are typically developed for manufacturing testing. As a result, extra efforts needed to deploy the solution are minimized.

### 3.1.1 Description of Board-Level Self-Test

Let us consider a PCBA (Fig. 3) which has an FPGA, a CPU and any other complex digital device (e.g., memories). Typically, all such devices incorporate IEEE 1149.1 JTAG/Boundary Scan facilities [36] that allow application of board-level tests. These tests are typically applied by means of an external test controller that is attached to an on-board JTAG connector. The test stimuli are shifted into the JTAG scan-chain via the Test Data Input (TDI) port and are captured on the Test Data Output (TDO) port. Two more signals – Test Mode Select (TMS) and Test Clock (TCK) are used to control JTAG test application. Each BS-capable device incorporates Boundary Scan Register (BSR) that becomes a part of JTAG scan-chain and is used to drive test stimuli into board interconnect and capture responses [37].

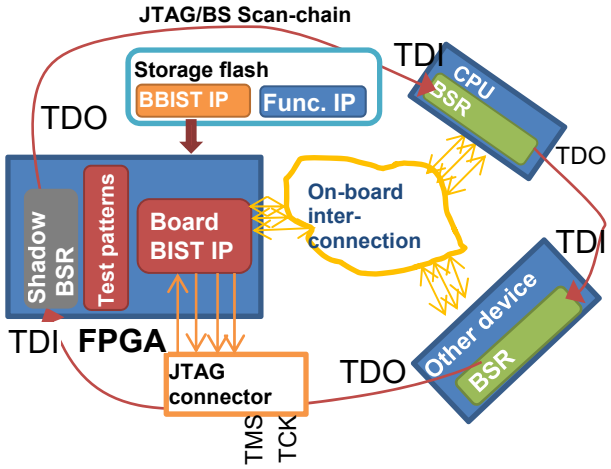


**Fig. 3 Boundary-Scan Board Built-in Self-Test**

Normally, BS tests are carried out only at the production site. After the board is manufactured, the JTAG infrastructure is rarely used and typically remains in idle state. However, it still exists and can be utilized for in-field board testing of in the way described below.

When board is powered up in the field, the FPGA device becomes initially configured with a BBIST IP which performs various board self-test procedures. Once tests are finished with no errors, the FPGA gets re-loaded and the start-up sequence continues. Both types of IPs (test and functional) can be stored in a single configuration storage device (e.g., Xilinx Configuration Flash or SPI flash). This scheme ensures small overhead, since no extra devices have to be mounted on board. However, the storage flash should be large enough to accommodate several firmware images.

A small amount of on-board DFT structures has to be added to the PCBA in order to support the proposed test technique. In total, 4 extra pins of the FPGA have to be reserved for BBIST IP and 4 extra nets on the PCB have to be routed. Note, that the abovementioned changes do not block the capability of making normal BS/JTAG test afterwards. This is especially useful in case the failure on PCBA is detected in-field and the failed product is returned to manufacturer.

The goal of the self-test is to check the connections between BS-capable ICs on board. In particular, the technique can execute typical BS tests and detect opens/shorts on interconnection lines. In certain cases, the faults on the interconnections between BS-capable and non-BS devices (e.g., RAM, Flash, sensors) can also be detected. The same BS routines that had been prepared for manufacturing test can be utilized by the BBIST IP. Despite some conversion may be required to adapt them, this conversion can be done automatically.

More information about architecture of BBIST IP and the procedure of board-level power-on self-test is provided in Deliverable D4.1.

### 3.1.2 Enhanced BBIST IP

BBIST IP described in the previous section is capable to apply Boundary Scan tests in field. However the test application speed is rather low due to the use of on-board scan-chains. The latter limits the applicability of BBIST IP by static test domain. However, it is also possible to overcome this restriction and extend BBIST IP towards support of high-speed and at-speed testing.

The idea is to combine BBIST IP with embedded instruments that are capable to test peripheral connected to FPGA. For this purpose, we can use already developed instruments such as at-speed memory test IP, frequency counter IP, instrument for bit-error rate test and others. As the same instruments are utilized during manufacturing test thus only small amount of effort is needed to embed them as a part of POST.

At the same time, embedded instruments in conjunction with BBIST greatly extend capabilities of the proposed POST beyond Boundary Scan interconnect test. For example, at-speed self-test procedures will be able to detect of timing-related faults and check the integrity of high-speed communication channels.

The schematic representation of enhanced BBIST IP is shown below (Fig 4).

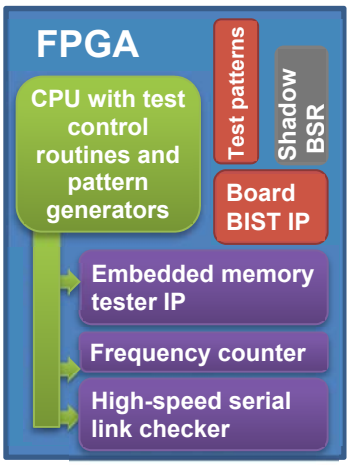


**Fig. 4 Enhanced Built-in Self-Test IP**

The picture above represents BBIST IP architecture (described in Section 3.1.1) complemented with a set of embedded test instruments. These instruments are controlled by test software which is running on a CPU located inside BBIST IP. The CPU controls embedded instrumentation and also is used to generate proper test patterns. For example, in case of usage DDR memory tester the CPU should be loaded with firmware capable for generation of at-speed DDR test patterns.

After infrastructure self-test described in Section 3.1.1 is finished, the CPU takes control over BBIST IP and initiates instrument-assisted test procedure. Depending on test software, CPU may allow multiple instruments run concurrently in order to minimize overall test time. At the same time, tests that require high-speed data exchange (memory test) should be run exclusively in order to achieve at-speed test pattern application rate.

### 3.1.3 Experimental results

In order to check the feasibility and cost of the proposed approach we applied it on a sample case. In particular, the experimental BBIST IP has been developed for a small Xilinx Spartan 3 FPGA (xc3s200tq144) and verified on a PCBA that contains two BS devices connected in a single JTAG scan-chain. First, the interconnection test was generated using BS test development software. In total 19 test patterns have been generated. Then, the generated test patterns were converted into the format suitable for BBIST IP (which resulted in 11,875 bits of test sequence). Experiments have shown that 445 LUTs are required for accommodation of the BBIST IP (which constitutes 11% of available resources). The amount of required memory depends on the length of the test sequence. For the generated test a single BRAM block was used (9% of total memory available).

## *3.2 Power-on self-test using debug features*

One of the aims of the work described in this report is to investigate the usage for test purposes of the debug features existing in several processors for supporting the software development. We will consider the ARM processors as a reference.

### 3.2.1 Background on the ARM debug features

This section uses Zynq-7000 SoC by Xilinx as a platform for concepts evaluation; therefore, we purposefully review debug and trace facilities in this device (see Fig. 5 for the general architecture). In particular, the Zynq-7000 conforms to the *ARM*

*CoreSight On-chip Trace and Debug Architecture* [30], which is not only a part of Zynq-7000, but is also widely adopted by various chip vendors in devices based on ARM Cortex-A, Cortex-M and Cortex-R cores.

According to the CoreSight Architecture, all the CoreSight components belong to one of the four classes: access and control, trace source, trace link and trace sink. Every configurable CoreSight component contains a set of memory-mapped registers that are accessible from an external debug equipment and from the application software. The block diagram of the CoreSight System implementation in the Zynq-7000 SoC is shown in Fig. 5.

#### 3.2.1.1 Access and Control Components

Components belonging to this class provide user access to the configuration of the CoreSight system, allowing him/her to decide, for example, the source of trace information and the communication mechanism among the components.

The ARM Debug Access Port (DAP) provides access to the internal memory and memory-mapped configuration registers of the device controllers via JTAG port, including the CoreSight components. In addition to the memory access port (MEM-AP) DAP may also include the ARM Peripheral Bus (APB) access port (APB-AP) and the Advanced High-performance Bus access port (AHB-AP). In case of Zynq-7000 SoC APB-AP enables access to peripheral in the FPGA part as an APB slave component. Furthermore, DAP implements a debug communication channel (DCC) that allows data exchange between the external test equipment and software running on the processor.

The Embedded Cross Trigger (ECT) component implements a mechanism to pass debug events between processor cores (hard and soft cores in case of Zynq-7000 SoC) allowing one CoreSight component to communicate with others via trigger events.

#### 3.2.1.2 Trace Source Components

Trace information can be generated by different components, representing different aspects of the system. All the trace source components are the masters of the AMBA trace bus (ATB).

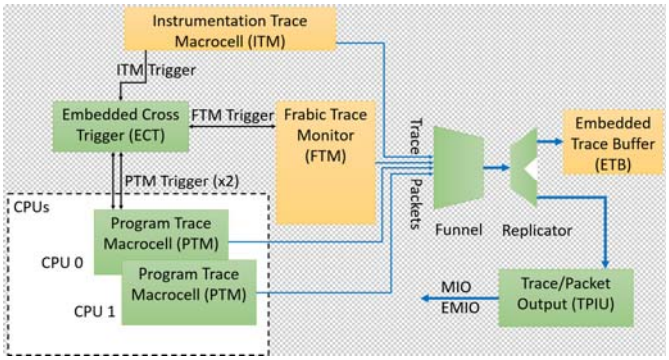


**Fig. 5 CoreSight System Diagram in Zynq-7000**

The Program Trace Macrocell (PTM) generates trace data containing information of the software running on the processor. There are two PTM instances in the Zynq-7000 due to its dual processor core architecture.

Instrumentation Trace Macrocell (ITM) on the other hand allows software developer to explicitly insert trace points into the software to ease the effort for application-level trace and debug (e.g., print type debug).

Fabric Trace Monitor (FTM) is a Xilinx specific trace Macrocell that complies with CoreSight architecture specification. FTM enables unified to ITM and PTM trace data generation by peripherals implemented inside the programmable logic (FPGA) of the Zynq-7000 SoC.

#### 3.2.1.3 Trace Link Components

Between the trace source and sink components there are two trace link components (*Funnel* and *Replicator*).

The ATB *Funnel* is the component for merging trace data from multiple sources (PTM, FTM and ITM) into a single stream and sending it to the ATB bus. The *Funnel* has an internal arbiter that users can tune to assign priorities to trace sources to be merged.

The ATB *Replicator* duplicates the trace stream onto its two output ATB master ports from the input ATB slave port. The *Replicator* is not a configurable component, thus, it is invisible to the user and has no memory-mapped registers. In Zynq-7000 device the *Replicator* output ports are connected to the Embedded Trace Buffer (ETB) and Trace/Packet Output (Trace Port Interface Unit, or TPIU).

#### 3.2.1.4 Trace Sink Components

The ETB and TPIU are implemented in Zynq-7000 as trace sink components. The ETB is an on-chip storage module with limited size (4KB) to enable short-window real-time and full-speed tracing. Meanwhile, the TPIU allows the trace packages to be output to the FPGA part or to the chip output pins. For the purpose of this work we used the first solution, and implemented a peripheral in the FPGA part connected to the TPIU's output via the Extended Multiplexed I/O (EMIO), using the peripheral to process the trace packages.

### 3.2.2 Monitoring IP

As described above, the CoreSight trace and debug infrastructure implemented in Zynq-7000 allows us to use a custom IP as a peripheral connected to the TPIU for monitoring the trace packages generated by different sources (Fig. 6).

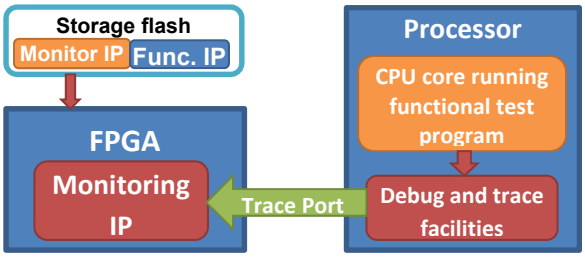


**Fig. 6 FPGA-monitored processor-based functional test**

To demonstrate the feasibility of the approach described in this sub-section, we implemented a simple monitoring IP to be mapped on the Zynq-7000 device. With the monitoring IP, we are able to extract on-the-fly the trace packages from the PTM and

read several relevant information about the test program execution including, for example, taken/not-taken decision, the target PC address of branch instructions, cycle accurate information between two branch instructions, the exception status of the processor.

To determine if the test running on the processor is executed correctly, we use the monitoring IP to directly compress the information extracted from the TPIU by sending them to a Multiple Input Shift Register (MISR) for signature computation. At the end of the test, the test program itself checks the signature generated by the MISR and compares it with the expected signature.

Note that the trace packages generated by PTM are grouped into several different categories: among them there are packages containing timestamp information that varies along the time, so a filter is implemented in the monitoring IP to remove these packages to make the signature obtained by the MISR deterministic. The structure of the monitoring IP is illustrated in the Fig. 7 and the resources consumption in terms of Look-Up Tables (LUTs) in the FPGA is reported in Table VII. The user can easily see that the amount of FPGA resources is quite limited. Moreover, it should be recalled that these resources are not subtracted from those that can be used by the application, since POST is performed when the FPGA has not been uploaded with the functional circuit, yet.

We also implemented an ARM Advanced eXtensible Interface (AXI) slave containing several registers which hold the configuration data and run-time status of the other components in the monitoring IP. Through these registers, the user can configure the monitoring IP at the beginning and check the status and final results of the MISR at the end of the test execution. The whole workflow in this demo is shown in Fig. 8.

The monitoring IP in the current version is still simple. However, it shows the feasibility of implementing such a peripheral to extract information from the processor in a non-intrusive way with the help of the existing debug and trace infrastructure. Possible extensions are possible, as mentioned in the Conclusions.

### 3.2.3 Evaluating the observability increase

In this section, we will present some experimental figures aimed at assessing the fault detection increase that one can achieve by adopting the observation mechanism supported by the debug features.

We considered a MIPS-like processor based on the RT-level VHDL description available at [18]. This processor is a 32 bits core composed of:

- A pipeline with 5 stages: instruction fetch, instruction decoding, execution, memory access, and registers update
- 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, implementing a Branch History Table.

The VHDL description was synthesized using the Synopsys Design Compiler with a technology library developed in-house. We used a test program manually developed by a test engineer knowing the netlist of the processor and targeting the maximization of the stuck-at fault coverage. The size and duration of this test program are 1,520 bytes and 14,617 clock cycles, respectively. Several fault simulation experiments

aimed at assessing the Fault Coverage achievable with such a test program with different observation mechanisms were performed using Synopsys TetraMAX. A programmable VHDL wrapper was also developed to support the computation of the Fault Coverage achievable with the different mechanisms, including the one corresponding to the debug features.

TABLE VII RESOURCE CONSUMPTION BY THE MONITORING IP

| Resource | | # Used by monitoring IP | % Used by monitoring IP |
|---|---|---|---|
| **Slice LUTs** | LUTs as Logic | 981 | 5.58 |
| | LUTs as Memory | 0 | 0 |
| **Slice Registers** | FlipFlops | 1,129 | 3.21 |
| | Latches | 0 | 0 |
| **Muxes** | F7 Muxes | 64 | 0.80 |
| | F8 Muxes | 32 | 0.70 |

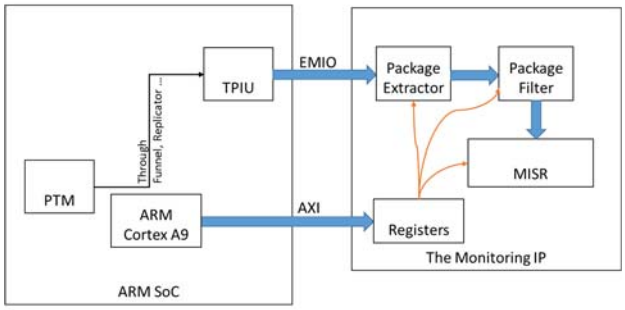


**Fig. 7 Diagram of the Monitoring IP**

Table VIII reports for the adopted processor the number of detected faults (out of the total of 268,424 stuck-at ones in the whole processor) using several observation mechanisms:

- Always observing all the output signals of the processor (*ATE*); this mechanism is the one which is typically adopted when performing end-of-manufacturing test; in this case, the device is mounted on an ATE, which provides full controllability of all input signals, and full observability of all output signals;
- Observing the content of the memory at the end of the test (*mem*); this mechanism is the one which is typically adopted for in-field functional test; the processor executes a test program, and then the final content of the memory area where results are written is checked for compliance with the expected values; clearly, this mechanism achieves lower Fault Coverage figures than the previous, due to the more limited observability we have during in-field test;
- Observing the debug port when a jump is executed (*dbgm_branch*); this mechanism is the one described in this work, using the debug features as available in the ARM processor.

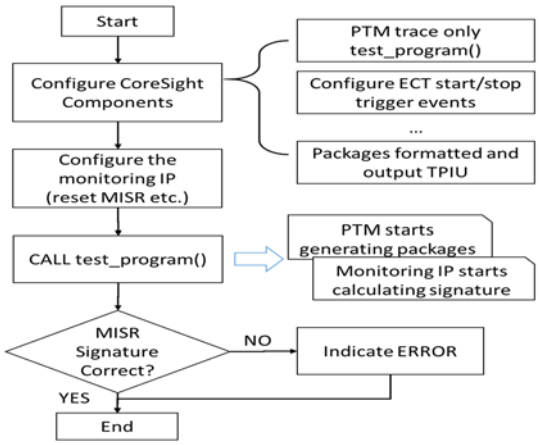


**Fig. 8 Workflow of the proposed system**

Table VIII EXPERIMENTAL RESULTS ON THE MIPS-LIKE PROCESSOR

| | ATE | mem | dbgm_branch | mem+ dbgm_branch |
|---|---|---|---|---|
| Detected | 240,591 | 218,787 | 48,083 | 240,547 |
| FC (%) | 90.30% | 82.12% | 18.05% | 90.29% |

It is interesting to note that the *dbgm_branch* mechanism proposed in this work, when combined with the *mem* one, allows a significant increase in the achievable Fault Coverage. In fact, a significant percentage of the faults detected by *dbgm_branch* are not detected by *mem*. Hence, the total number of faults combining the two mechanisms (reported in the rightest column) is higher than both of them, and even higher than what we can observe with the first mechanism, which is not usable in in-field test. This result can be explained by recalling that the *dbgm_branch* mechanism allows accessing information about the internal behavior of the processor.

It is finally worth noting that a non-negligible percentage of the faults in the processor (2,429 according to TetraMAX, at least 3,291 according to [31]) are untestable, and thus never produce any misbehavior. Hence, the Testable Fault Coverage figures are definitely higher than the reported ones.

## 3.3 Conclusions

In this section we proposed a comprehensive architecture for power-on PCBA test. This architecture assumes that an FPGA module exists in the system, and uses it during POST to implement some IPs in charge of performing different kinds of testing.

First, we propose a solution able to support the easy re-use of Boundary Scan patterns developed for end-of-production test (e.g., to test interconnections among components on the board). The solution can be adopted for power-on test and uses the available FPGA. Its major advantages are the limited cost for its adoption and its limited intrusiveness into the whole design flow.

Secondly, we discuss the usage of some debug features originally introduced in processors to support software development in order to increase the observability (and hence the fault and defect coverage) during the functional test of PCBAs, with special emphasis on in-field test. In particular, these features allow the on-the-fly access to trace information during the test execution that can be monitored using a special hardware module that may be mapped on an external FPGA. We demonstrated the feasibility and cost of the solution on a Zynq-7000 system by Xilinx equipped with an ARM core. To assess the increase in Fault Coverage we mimicked it on a MIPS-like

processor whole gate-level netlist is available, and showed that the proposed technique can achieve figures even higher than those obtained during end-of-production testing, when the processor can be tested using an ATE. In this way, the defect coverage that can be obtained during in-field test of a PCBA is greatly enhanced, thus successfully attacking important issues, such as the reduction of NFF.

# 4 DTA Functional Test Support

The state of the art test for mixed-signal blocks is to test them functionally. Even though several BIST solutions have been proposed, they are seldom used in production test. Unfortunately, functional test is an expensive procedure: precise stimuli sources are needed, these sources require long settling times to reach the needed precision, and the circuit has to undergo a multitude of tests to check every value defined in the specification.
Means of optimizing the functional tests is to remove the need for precise stimuli or even to test the mixed-signal blocks without resorting to functional tests altogether.
Using a Successive-Approximation-Register Analog to Digital Converter (SAR-ADC) [24] as an example, this report describes one possible approach at how to optimize the functional test of such an Analog to Digital Converter (ADC).

## *4.1 Rationale and general overview*

The purpose of mixed-signal production testing is to weed out circuits that do not meet specifications because of production defects. This poses a challenge to test optimization techniques because what is needed is to reduce the test effort while keeping the delivery quality unchanged. This means no test escapes (bad circuits that are labeled as "good" by the test) and no yield reduction (but good circuits that are labeled as "bad" by the test are allowed).
In this report we will use capacitive SAR-ADCs as an example on how to optimize production tests without compromising quality.

## *4.2 Conventional test procedure and its shortcomings*

During conventional production test, a precise voltage is applied to the ADC inputs, then a conversion is initiated, and the ADC's conversion result is recorded. The input voltage is incremented, and the conversion and result storing cycle is repeated until the input range of the ADC is swept through. This procedure amounts to applying a ramp at the input of the ADC and recording the conversion results continuously, and subsequently storing them into memory. After post processing of the ADC output codes, the values representing the ADC performance can be calculated: Offset, gain, Integral Non Linearity (INL), Differential Non-Linearity (DNL), etc. As it can be seen from the description, the measurement is quite lengthy.
Moreover, the tests are conducted in a production test environment, which is not noise free. To reduce noise influences, conversions have to be made several times at different input voltages. This alleviates the noise effect through averaging of the ADC conversion values. Conclusively, optimizing these tests is worth the effort.

To summarize, the shortcomings associated with ADC functional test are

- Precise stimuli have to be applied to the ADC inputs
  - Provided by expensive and specialized ATE sources, leading to less "degrees of freedom" in the choice of ATE
  - Precision sources need long settling times, increases test time
  - Stimuli traces are sensitive to noise and need to be shielded, this crowds the load board, and reduces the number of chips that can be tested in parallel
- The test procedure implies long testing times
  - High number of measurements to check compliance to specification
  - All values within the input range have to be checked

- Averaging of numerous measurements is needed because of environmental noise.

## 4.3 Implementing a structural test to reduce test efforts

The shortcomings described above actually define what has to be done. Our main goals are:

- Detect all defects: Stimulate all analog parts
- Save costs: Test faster and without precision stimuli.

These contradictory goals lead to questioning the status-quo of the conventional production testing of ADCs. In particular, the following issues should be faced:

- Is a full ramp covering the whole input range actually needed?
- Is an external signal actually needed to test the functionality of the ADC?
- Can the underlying principle of the capacitive SAR-ADC help making the measurements?
- Are there "values" within the ADC that can be used to compute INL and DNL?

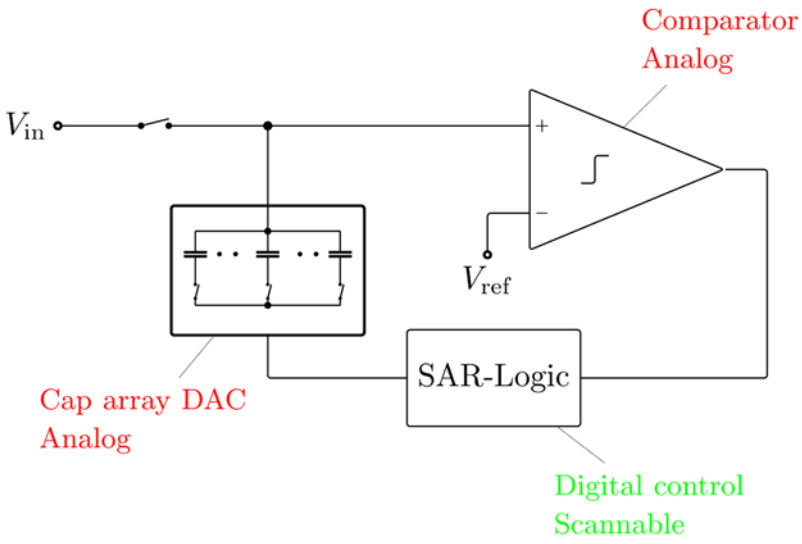


**Fig. 9 Major blocks in a capacitive SAR-ADC**

To gain insight, let us have a look at the SAR-ADC as a system (see Fig. 9) which shows the main blocks in a capacitive SAR-ADC. It consists of a comparator, a DAC, and a digital logic part. The digital logic controls the switch settings of the DAC and stores the conversion results of the ADC. If the logic is presumed to be fully scannable, then there is no need to consider detecting defects in it, as defects would be found during scan test. What remains are the red marked analog blocks: comparator, and cap-array DAC. This means that the only parts of the ADC where we have to detect defects are the comparator and the DAC.
Recall that INL and DNL are the values that cause the need for a ramp test with high precision stimuli sources. If we take into account that these ADC specification values are defined by the capacitance ratios of the DAC, then we can deduce that we could compute these values if a measurement of the capacitance ratios were possible.

The goals stated above (detect all defects and save costs) boil down to measuring the capacitances (ratios) using the comparator and the capacitance array of the DAC.

### 4.3.1 Measuring a capacitance

There are different methods to measure the value of a capacitance. One such method is to charge the capacitance with a known current, and measure the time needed to reach a certain voltage. Another method would be to charge the capacitance to a voltage and then use a known resistor to discharge it. In both methods the time needed to reach a certain voltage is used to compute the capacitance value.
For both methods we see that at *some point of* the measurement, we need a value that has to be known precisely in order to be able to compute the capacitance value. Unfortunately, due to process variations, integrated circuit elements have widely varying parameters. In this case the only means to provide a precisely known dimension would be to have it provided as an input from the ATE, which gets us back to providing precise input stimuli.
Nevertheless, let us look at a capacitance measurement that *could be done* inside an integrated circuit. Because the underlying principle might help in finding alternatives that would make the measurement possible.
Recapitulating the measurement methods mentioned above, the key proposed idea is to charge or discharge the capacitance and measure the time needed to do so.
Using a current can be ruled out. Currents of unrealistically low magnitudes would be needed in order to charge the small capacitances used in the ADC, and still being able to measure the time needed for the voltage to reach a certain value. Using a resistor seems to be a viable method. However, the resistance could be such that the resistor would need a substantial amount of layout area, making the method not feasible. A better solution could be to use a switched capacitance.

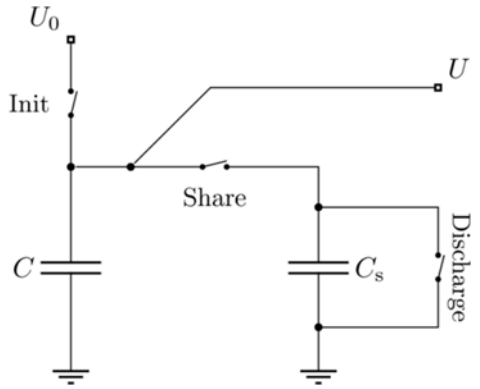


**Fig. 10 Discharging a capacitance with a switched capacitor**

Consider the capacitance C in Fig. 10 as being the capacitance to be measured.
As an initial step, the capacitance C is charged to a voltage $U_0$. Now we repeatedly alternate closing the switches “Share” and “Discharge”. The capacitance $C_s$ repeatedly “leaks” charge from C, which reduces the voltage across C’s terminals. What a resistor would have been doing continuously, namely “leaking charge”, is done by the capacitance in discrete packets. On average, the effect is the same. That is the essence of switched capacitor circuits.
After $n$ of these discharge cycles, the capacitance voltage is:

$$U = U_0(\frac{C}{C + C_s})^n$$

The equation tells us that if we manage to measure precisely the voltage reached after $n$ discharge cycles, then we would know the value of $\frac{C}{C+C_s}$. The capacitance we would like to measure being C, this implies that we have to know the exact value of $U_0$ and that of $C_s$. This seems not to be what we want, because what we know from integrated circuits is that no absolute value is known precisely.
To gain more insight, we inspect the whole proposed measurement system:

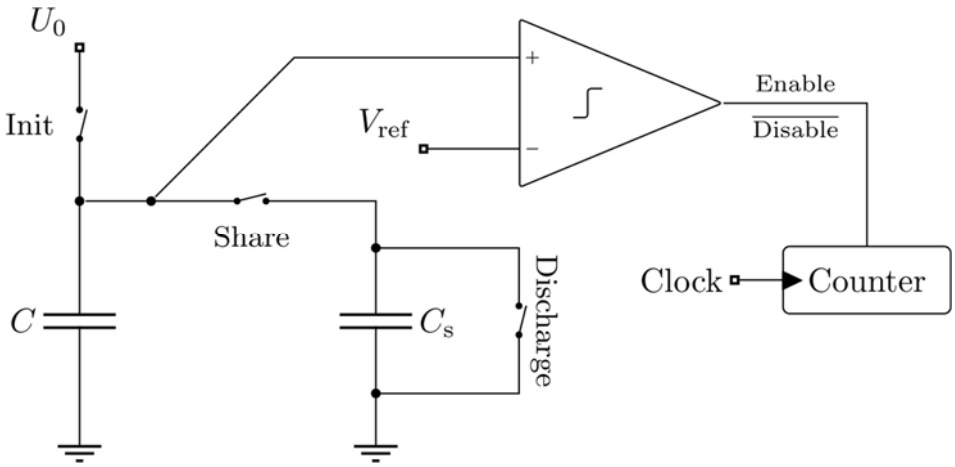


**Fig. 11 Measuring the discharge time of a capacitance**

In Fig. 11 a comparator and a clock counter were added to the circuit of Fig. 10. Each clock tick increments the counter. It corresponds to one discharge cycle of the capacitance C. As long as the voltage of the capacitance we want to measure is above $V_{ref}$ the counter keeps counting. As soon as the reference voltage is reached, the comparator disables the counter, freezing the number of discharge cycles that were needed to reach $V_{ref}$. Looking back at the goals we defined in 4.3, we can see that if we use the ADC's comparator we already accomplished half of the task. The ADC's comparator is now part of the measurement system and any of its defects would lead to a measurement error that can be readily detected.

Let us have a look at what the contents of the clock counter mean.

If we assume that the comparator is not ideal, and take its offset into account, we can write:

$$U_0(\frac{C}{C + C_s})^n = V_{ref} + V_{offs}$$

We can see that an additional value, $V_{offs}$, appears in the equation. Neither we can control its value nor we know its value precisely. Actually, the values we need to compute the capacitance C from the above equation are either unknown or not known precisely: measuring the value of the capacitance will not be possible.
But in section 4.3 our aim was to either measure the capacitances of the ADC or their *ratios*.

If the absolute value of the capacitance is out of reach, could the measurement of the ratios still be possible? To assess this we need to look at the whole ADC.

### 4.3.2 Measuring the capacitance ratios of the ADC

Fig. 12 shows an example ADC that would undergo capacitance or capacitance ratio measurements.

The difference with respect to Fig. 11 is that now the ADC-switches $S_{\{0,1,2,3\}}$ are repurposed to choose which capacitance is to be measured. Each switch allows connecting one of the ADC capacitances to the measurement network.

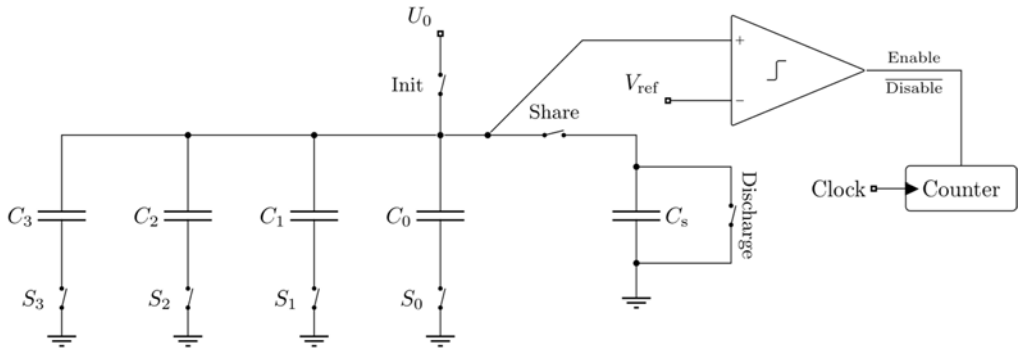


**Fig. 12 Example of a 3-bit ADC with capacitance measurement**

For each possible switch setting $S_3S_2S_1S_0$ there is a capacitance value connected to the measurement network. For switch setting $i$ we have:

$$U_0(\frac{C_i}{C_i + C_s})^{n_i} = V_{ref} + V_{offs}$$

The setting with all switches off is not considered as a valid configuration, as there would be nothing to measure. A more interesting setting is the one where all switches are on. This allows measuring the global capacitance $C_g = C_3 + C_2 + C_1 + C_0$. The value of $C_g$ is needed to compute the capacitance ratios in order to derive INL and DNL (The ratios define the voltage steps at which the ADC output codes change).

This switch setting with all capacitances connected results in:

$$U_0(\frac{C_g}{C_g + C_s})^{n_g} = V_{ref} + V_{offs}$$

If we just keep $C_3$ in, and disconnect all other capacitances, we get:

$$U_0(\frac{C_3}{C_3 + C_s})^{n_3} = V_{ref} + V_{offs}$$

Both equations have the same right side; we can equate the left sides:

$$U_0(\frac{C_3}{C_3 + C_s})^{n_3} = U_0(\frac{C_g}{C_g + C_s})^{n_g}$$

Leading to:

$$(\frac{C_3}{C_3 + C_s})^{n_3} = (\frac{C_g}{C_g + C_s})^{n_g}$$

After this step we can see that three values that were causing the measurement of the capacitance values to be impossible have just disappeared from the equation.

This implies that the values $V_{ref}$, $V_{offs}$, and $U_0$ are of no importance *as long as they do not change during the measurement*.

Apart from the capacitance values we want to measure, the only unknown value remaining in the equations is the value of the discharge capacitance $C_s$.

As we are dealing with positive values, the equation can be rearranged as follows:

$$\left(\frac{C_3 + C_s}{C_3}\right)^{n_3} = \left(\frac{C_g + C_s}{C_g}\right)^{n_g}$$

This leads to:

$$\left(1 + \frac{C_s}{C_3}\right)^{n_3} = \left(1 + \frac{C_s}{C_g}\right)^{n_g}$$

Now if the capacitance $C_s$ were made to be much smaller than any of the capacitances to be measured, we could approximate the power terms:

$$1 + n_3 \frac{C_s}{C_3} \approx 1 + n_g \frac{C_s}{C_g}$$

Leading to:

$$\frac{C_3}{C_g} \approx \frac{n_3}{n_g}$$

*This is one of the capacitance ratios we need.*

After performing the same measurements with the remaining capacitances, we can compute the individual ratios needed to predict the INL and DNL of the ADC.

For every capacitance $C_i$ we have:

$$\frac{C_i}{C_g} \approx \frac{n_i}{n_g}$$

## *4.4 Experimental Results*

By adding two switches, a small capacitance and a counter, the capacitance ratios of an ADC can be measured. The need for precise input stimuli is removed for INL and DNL measurements. All circuit blocks of the ADC are used for the measurement; any defect occurring in one of the blocks will be detected. As the measurements occur as

a BIST (within the chip), and there is no need for any precise voltage or current, this measurement is not only fit for production test, but also for being activated in the field for Online Test purposes.

The discharge principle discussed above has been implemented within a test chip. In this section measurement results of the silicon are presented.
The switching network and the counter have been added to a 12 bit ADC. The implemented circuit differs from the theoretical one discussed above; this was needed in order not to impede the normal function of the ADC by the leakage currents that result from adding switches to the sensitive node at the input of the comparator.

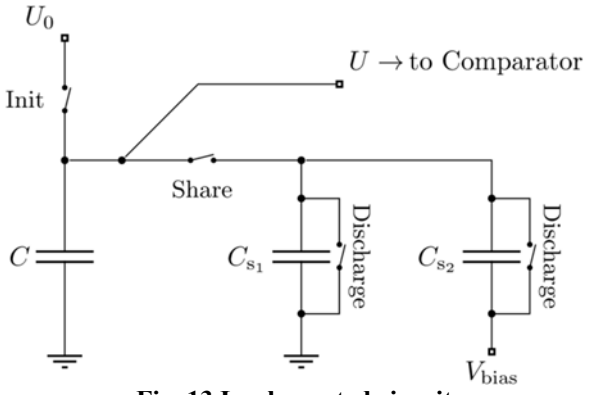


**Fig. 13 Implemented circuit**

The difference in the implemented circuitry leads to a slightly different equation for the computation of the capacitance ratios:

$$\frac{C_i}{C_g} \approx \frac{n_i}{n_g} + \frac{C_{s1+}C_{s2}}{C_g}\,\frac{n_i - n_g}{n_g}$$

As a first check that the discharge network is operating as expected, a measurement for each code of the switch network was conducted. It can be seen that all measurements lead to valid values (e.g., the counter works, and doesn't stick to zero or to its maximum value). Even the linear deviation predicted by the altered equation is visible in the right part of the blue graph. The bowing back in the left part can be attributed to charge injection effects, as the used capacitances were very small.

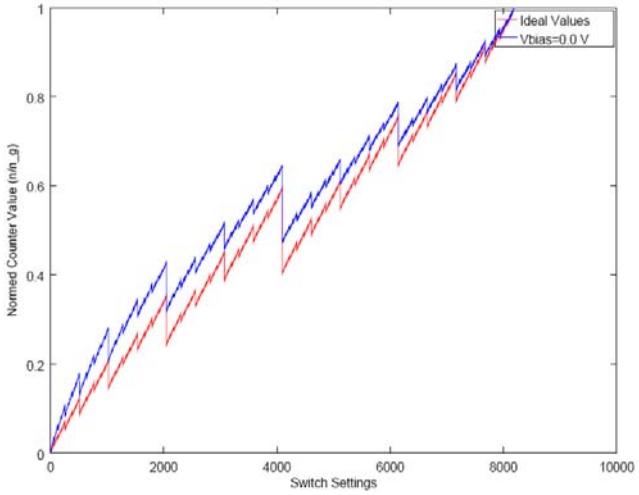


**Fig. 14 Measurement of all codes of the switch settings**

In order to make sure that the discharge capacitances comply with $C_s \ll C_i$, the computations were not performed on values where only one capacitor was switched into the measurement network. Instead we used values that resulted from *leaving out* one capacitor from the network. This ensured that enough capacitance was always present in the measurement in order to comply with $C_s \ll C_i$. Instead of computing the capacitance directly, an equation system had to be solved.

The ratio equation has a term containing the discharge capacitances: $\frac{C_{s1}+C_{s2}}{C_g}$. We have to find its value in order to compensate for it. In order to approximate the needed value, both measurements with the highest amount of capacitance in the measurement network are assumed to be the most accurate, and are used to compute the value of the linear factor. With $C_g$ the total capacitance, and $C_1$ the capacitance leaving out the highest capacitance, and with $n_g$ and $n_1$ representing the respective counter values of the measurements, we can approximate the term as:

$$\frac{C_{s1}+C_{s2}}{C_g} \approx \left(\frac{C_1}{C_g} - \frac{n_1}{n_g}\right) \frac{n_g}{n_1 - n_g}$$

After correcting for the additional term and solving the resulting equation system, we obtain the capacitance ratios. Note that the capacitances shown in Table IX are:

- Normed so that the highest capacitance has no error
- Represent the number of "unit capacitances", with the lowest capacitance having a value of one unit.

Table IX Measurement Results

| Measured | Ideal | Comment |
|---|---|---|
| 824,00 | 824 | Considered error free |
| 493,81 | 496 | |
| 294,72 | 296 | |
| 174,65 | 176 | |
| 106,32 | 108 | |
| 66,16 | 64 | |
| 38,67 | 36 | |
| 23,61 | 22 | |
| 12,53 | 12 | |
| 6,01 | 7 | |
| 3,53 | 4 | |
| 2,20 | 2 | |
| 1,45 | 1 | Unit capacitance |

Table IX shows that the computed capacitance values (or ratios) closely agree with the ideal capacitance values with which the ADC was designed. Slight inaccuracies in the low valued capacitance values result from charge feed-through phenomena and are to be addressed in a further test chip.

## *4.5 Conclusions*

Chip-internal measurements of capacitance ratios are possible. Functional tests are not needed to test INL and DNL specifications for this kind of ADC. Further, as the measurements are done in BIST form, even online test becomes possible without the need for precise stimuli, which unburdens the customer's system.

Refinements have to be done to avoid extraneous terms in the measurements equations (a further test chip with refinements is being produced). Work is being done to obtain a linear discharge curve, instead of the power law discharge.

# 5 Conclusions

In this deliverable we have described a set of techniques intended to improve the quality and effectiveness of a functional test aimed at working on devices or boards. In this way we face one of the major issues of the BASTION project, i.e., improving the quality of the test (both end-of-line and in-field), thus reducing the impact of the NFF phenomenon.

The different techniques proposed in the document face different aspects of test.

The techniques outlined in Section 2 aims at reducing the size and duration of a functional test performed on devices (either at the end of manufacturing or in field). In this way, not only the cost for test is reduced, but in some cases (e.g., when it is applied in field), even its feasibility is affected. The adopted approach is a *static* one, i.e., compaction is performed once the test generation phase is completed. The computational cost of the compaction phase can become significant, and it is thus also important to tame it in order not to let it explode, while still guaranteeing that the achieved fault coverage remains at least unchanged.

The techniques described in Section 3 address the test of PCBAs. In this section the key idea is to use the programmable resources (e.g., FPGA devices) often existing in today's boards, and configure them suitably during the test phase (e.g., during the Power-On Self-Test), thus contributing to support a test which achieves a higher fault coverage and re-use vectors developed for the end-of-production test phase.

Finally, the solution proposed in Section 4 specifically address the test of a common type of Analog to Digital Converters, showing how to combine a functional approach with the usage of some limited DFT, thus contributing to improve the effectiveness of the test.

The content of this deliverable summarizes the main achievements of the BASTION project partners involved in Task 4.2.

# 6 References


[1] Big Trouble with "No Trouble Found" Returns, Accenture Report, 2008, Available at: http://www.accenture.com/SiteCollectionDocuments/PDF/Accenture_Returns_Repairs.pdf

[2] S. Thatte and J. Abraham, “Test Generation for Microprocessors”, IEEE Transactions on Computers, vol. 29, no. 6, pp. 429–441, June 1980.

[3] M. Psarakis et al., “Microprocessor Software-Based Self-Testing”, IEEE Design & Test of Computers, vol. 27, no. 3. May-June 2010, pp. 4-19.

[4] A. Jutman, M. Sonza Reorda, H.-T. Wunderlich, “High Quality System Level Test and Diagnosis”, IEEE Asian Test Symposium, 2014.

[5] A. Riefert et al., “An effective approach to automatic functional processor test generation for small-delay faults”, Proceedings of the Conf. on Design, Automation and Test in Europe (DATE), 2014.

[6] P. Bernardi et al., “An Effective technique for the Automatic Generation of Diagnosis-oriented Programs for Processor Cores”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, pp. 570-574, 2008.

[7] E. Sánchez, M. Schillaci, G. Squillero, “Enhanced Test Program Compaction Using Genetic Programming”, 2006 IEEE Congress on Evolutionary Computation, pp. 865-870, 2006

[8] I. Pomeranz and S. M. Reddy, "On Static Compaction of Test Sequences for Synchronous Sequential Circuits", in Proc. Design Autom. Conf., 1996, pp. 215-220.

[9] M. S. Hsiao, E. M. Rudnick and J. H. Patel, "Fast Algorithms for Static Compaction of Sequential Circuit Test Vectors", in Proc. VLSI Test Symp., 1997, pp. 188-195.

[10] I. Pomeranz and S. M. Reddy, "Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential Circuits", in Proc. Intl. Conf. on Computer Design, 1997, pp. 360-365.

[11] M. S. Hsiao and S. T. Chakradhar, "State Relaxation Based Subsequence Removal for Fast Static Compaction in Sequential Circuits", in Proc. Design Autom. and Test in Europe, 1998, pp. 577-582.

[12] S. K. Bommu, S. T. Chakradhar and K. B. Doreswamy, "Static Compaction using Overlapped Restoration and Segment Pruning", in Proc. Intl. Conf. on Computer-Aided Design, 1998, pp. 140-146.

[13] X. Lin, W.-T. Cheng, I. Pomeranz and S. M. Reddy, "SIFAR: Static Test Compaction for Synchronous Sequential Circuits Based on Single Fault Restoration", in Proc. VLSI Test Symp., 2000, pp. 205-212.

[14] I. Pomeranz and S. M. Reddy, "Vector Replacement to Improve Static Test Compaction for Synchronous Sequential Circuits", IEEE Trans. on Computer-Aided Design, Feb. 2001, pp. 336-342.

[15] I. Pomeranz and S. M. Reddy, "Enumeration of Test Sequences in Increasing Chronological Order to Improve the Levels of Compaction Achieved by Vector Omission", IEEE Trans. on Computers, July 2002, pp. 866-872.

[16] I. Pomeranz and S. M. Reddy, "Vector Restoration Based Static Compaction using Random Initial Omission", IEEE Trans. on Computer-Aided Design, Nov. 2004, pp. 1587-1592.

[17] J.G. Tong, M. Boulé, Z. Zilic, “Test compaction techniques for assertion-based test generation”, ACM Trans. on Design Automation of Electronic Systems (TODAES), December 2013, pp. 1-29.

[18] "miniMIPS Overview," opencores.org, 2009. [Online]. Available at http://opencores.org/project,minimips.

[19] "NanGate FreePDK45 Generic Open Cell Library", [Online]. Available at https://www.si2.org/openeda.si2.org/projects/nangatelib.

[20] P. Bernardi et al., “On the Functional Test of the Register Forwarding and Pipeline Interlocking Unit in Pipelined Processors”, Proc. of the 14th International Workshop on Microprocessor Test and Verification (MTV), 2013, pp. 52-57

[21] P. Bernardi et al., “On the in-Field Functional Testing of Decode Units in Pipelined RISC Processors”, Proc. of the IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014, pp. 298-303

[22] E. Sanchez, M. Schillaci, G. Squillero, Evolutionary Optimization: the μGP toolkit, Springer, 2011

[23] E. B. Eichelberger and T. W. Williams, “A logic design structure for LSI testability”, Proc. of Design Automation Conference (DAC), 1977, pp. 462–468.

[24] J.L. McCreary and P.R. Gray, ”All-MOS Charge Redistribution Analog- to-Digital Conversion Techniques – Part 1,” IEEE Journal of Solid State Circuits,vol. SC -10, no.6, 1975, pp.371-379.

[25] Sanyal, A.; Chakrabarty, K.; Yilmaz, M.; Fujiwara, H. “RT-level design-for-testability and expansion of functional test sequences for enhanced defect coverage,” IEEE Int Test Conf (ITC’2010), pp. 1-10.

[26] A. Jutman “Filling a Gap in Board-Level At-Speed Test Coverage”, IEEE Int. Workshop on Defects, Adaptive Test, Yield and Data Analysis (DATA’2015), pp. 1-7.

[27] Fang, H.; Wang, Z.; Gu, X.; Chakrabarty, K. “Mimicking of Functional State Space with Structural Tests for the Diagnosis of Board-Level Functional Failures”, IEEE Asian Test Symp. (ATS’2010), pp. 421-428.

[28] I. Aleksejev, A. Jutman, S. Devadze, S. Odintsov and T. Wenzel, "FPGA-Based Synthetic Instrumentation for Board Test," IEEE International Test Conference, ITC’2012.

[29] J. Perez Acle, R. Cantoro, A.T. Hailemichael, E. Sanchez, M. Sonza Reorda, Observability solutions for in-field functional test of processor-

based systems, XXX IEEE Conference on Design of Circuits and Integrated Systems (DCIS), 2015

[30] http://www.arm.com/products/system-ip/debug-trace/

[31] A. Riefert et al., “On the Automatic Generation of SBST Test Programs for In-Field Test”, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015

[32] Larsson, Erik; Eklow, Bill; Davidsson, Scott; Aitken, Rob; Jutman, Artur; Lotz, Christophe, “No Fault Found: The root cause”, IEEE 33rd VLSI Test Symposium (VTS), 2015

[33] Jasnetski, A.; Raik, J.; Tsertov, A.; Ubar, R., “New Fault Models and Self-Test Generation for Microprocessors Using High-Level Decision Diagrams”, IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2015

[34] Stroud, C.; Ping Chen; Konala, S.; Abramovici, M., “Evaluation of FPGA Resources for Built-In Self-Test of Programmable Logic Blocks”, ACM Fourth International Symposium on Field-Programmable Gate Arrays, 1996

[35] M. G. Gericota, G. R. Alves, M. L. Silva, J. M. Ferreira, "On-line Testing of FPGA Logic Blocks Using Active Replication", Norsk Informatikkonferanse (NIK'2002), Kongsberg, Norway, Nov. 2002, pp. 167-178.

[36] IEEE Standard test access port and boundary-scan architecture, IEEE Std. 1149.1-2001, 2001.

[37] K.P. Parker. The Boundary-Scan Handbook, Kluwer Academic Publishers, Boston, MA, USA, 2003, 373 pages

[38] H. Chen, “Adaptive System Level Test for High Volume Mobile Phone Chips”, IEEE Asian Test Symposium, 2015, Invited Talk