POLITECNICO DI TORINO

DOCTORATE SCHOOL
Ph.D. in Computer And Control Engineering – XXV cycle

PhD Thesis

New Techniques for Reliability Characterization of Electronic Circuits

Lyl M. Ciganda Brasca

Advisor
Prof. Paolo Bernardi

February 2013

Lovingly dedicated to my parents, Dominga and Yamandú, continuous source of inspiration, my "sweet rocks".

Acknowledgements

It is with immense gratitude that I acknowledge the support and help of my advisor, Paolo Bernardi, whom with great patience and sense of opportunity guided my work during these years.

I would also like to thank Professor Matteo Sonza Reorda for his time, encouragement, and expertise throughout this project.

I consider it an honour to have worked with Michelangelo Grosso and Ernesto Sanchez, thank you for the stimulating discussions and creative ideas.

Special thanks go to my fellow lab members at Politecnico di Torino: Alberto, Alessandro, Davide, Fabio, the four Marcos, Mauricio, Niccolò, Salvatore, and all the students who came and went during these years, you all made my work much more fun.

To the staff and students at TIMA Laboratoire RMS group (2012), I am grateful for the chance to visit and be a part of the lab. Thank you for welcoming me as a friend and helping to develop some of the ideas in this thesis.

I am indebted to my many colleagues from Montevideo, beautiful people and excellent professionals who continuously encouraged me in this path.

This thesis would have remained a dream had it not been for my husband, Pablo Scanniello, whose constant support, selfless generosity and brilliant mind provided me light at all times. The baby-to-be was also a stupendous source of strength and hope for the last mile.

To my siblings, Antonio, Rosario, Yamandú, Carolina, Diego and Verónica, and all my beautiful nephews, thank you for being a never ending source of joy and happiness. And to all my family members, especially the ones who are no longer with me, your lives were truly inspiring.

Last but not least, to all my girlfriends round the globe, near enough to share a coffee or far away, just able to chat and mail, thank you for your open ears (and hearts) and for your wise pieces of advice.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Integrated electronic systems are increasingly used in an wide number of applications and environments, ranging from critical missions to low cost consumer products. Information processing has been thoroughly integrated into everyday objects and activities, in the so-called ubiquitous computing paradigm. This wide distribution is caused mainly by the miniaturization of semiconductor devices (transistor channel length scaling from 180 nm in 1999 to 22 nm in 2012), which allows integrating a complete system on a single chip (SoC). However, there are many difficult challenges associated with continued cost reduction, size reduction, improved performance and improved power efficiency.

One of these challenges is the reliability of these electronic systems. Important research efforts are aimed at improving the reliability of semiconductors. Manufacturing processes, intrinsic aging phenomena of components and environmental stress may cause internal defects and damages during the lifetime of a system, possibly causing misbehaviours or failures. In order to guarantee product quality and consumer satisfaction, it is necessary not only to discover faults as soon as possible in the manufacturing process, but also to continuously check for their absence throughout a product lifetime.

Today's modern systems have become increasingly complex to design and build, while the demand for reliability and cost effective development continues. Reliability is one of the most important attributes in all these systems, including aerospace applications, real-time control, medical care, defence equipment, transportation, communication, entertainment products, agriculture, energy and environmental systems. Growing international competition has increased the need for all designers, managers, practitioners, scientists and engineers to ensure a high level of reliability of their product before release and during mission time, at the lowest cost. The interest in reliability has been growing in recent years and this trend will continue during the next decade and beyond.

With testers being expensive pieces of equipment and the cost of transistors continuously decreasing, it make sense to use some of these low-cost transistors to replace the costly test tools, whenever possible.

The first low cost approach we can think about is using the devices themselves to implement their own test. This is the underlying motivation of functional Software-Based Self-Test (SBST): a fast, powerful microprocessor, which has lots of resources, could certainty help in its testing procedure. Having the outstanding advantages of enabling at-speed testing, zero area overhead and actually testing the device's operation, this approach also has some drawbacks. Even if SBST is essentially suitable for online testing (and sometimes it is the only possible approach), it requires some dedicated system memory for the functional testing data, which can reach very big sizes. Also some faults happen to be functionally untestable; i.e., you cannot detect them exclusively by running proper software routines. For this reason a combination of both functional and structural test approaches is common practise.

A second natural approach to low cost testing is design for test (DfT). Add some extra (cheap) area on-chip specifically in charge of performing and managing tests. The DfT path started long ago, but it is still a key element in 2012 International Technology Roadmap for Semiconductors (ITRS)[**1**] test roadmap. Different sorts of DfT enable the use of low cost testers, contributing to the full checking of a device, and may also be reused for online testing purposes. Logic and Memory Built-In Self Test (BIST) schemas are usual practises. Analogue DfT, even if it is not as advances as digital one, is also an interesting strategy, especially when the analogue or mixed-signal device is integrated in a wider digital system like a SoC

Finally, there are some fields where the use of external (and generally expensive) testers is mandatory. Diagnosis is one of the cases in which an Automatic Test Equipment (ATE) is needed to store the huge amount of retrieved data and to drive the cyclic characteristic of the diagnosis procedure. In particular, even if memories are commonly tested making use of internal BIST structures, their diagnosis demands the use of a tester. Another interesting and blooming field is that of the mixed energy-domain devices as Micro Electro Mechanical Systems (MEMS). MEMS require unique testing apparatus applying both electrical and physical stimuli: movement, pressure, magnetic fields. Additionally, they not only need to be exhaustively tested but in most of the cases also calibrated.

The work described in this thesis falls in low cost testing domain. Strategies for new and/or improved SBST, DfT and ATE mechanisms are proposed, implemented and evaluated. The strategies deal mainly with memories, processor and mixed-signal devices (analogue-to-digital converters is our target device) embedded in Systems-on-a-Chip, where standard communication protocols and wrappers are used to communicate with the device under test. The thesis is organized as follows:

Chapter 2, “Reliability Characterization” gives some basic concepts and general background information. Different kinds of testing methodologies are described, including the equipments, tools and methods they used, and the results that can be obtained with each. Also some competent international standards are presented.

Chapter 3, “Characterization - Different devices, different strategies” describes characteristics and limitations in the testing of some specific electronic circuits and introduces different approaches available in the state of the art. In particular we describe the solutions for embedded microprocessors, memories, mixed signal devices and MEMS sensors.

Chapter 4, “Proposed test programs for Software-Based Self-Test of microprocessors” presents SBST approaches for three not-easy to test units within a microprocessor: Prediction unit, Address Adder unit and Data Forwarding & Pipeline Interlock unit. First, the behaviour and architecture of each module is described. Then, we discuss the methodology for generating SBST patterns. Finally, a complete test program is explained, implemented and experimented in academic and industrial case studies for all three units. Obtained results support the proposed strategies.

Chapter 5, “Proposed Infrastructure-IPs to enhance self-testing capabilities of digital devices” introduces a new in-chip testing concept, which enables faster and cheaper testing of embedded memories and microprocessors. Two digital designs implementations are described, both of which are particularly suitable for online testing. Experimental results on real case studies are reported for the two presented designs.

Chapter 6, “Proposed enhanced Automatic Test Equipment – can we make it better, faster, stronger?” displays a methodology and suitable hardware implementation of a digital low cost tester. The approach benefits from FPGAs flexibility and is presented both for SoC embedded memories diagnosis and MEMS inertial sensor testing and calibration. Background on both fields is provided. Results obtained in real case experiments are shown.

Chapter 7, “Conclusions” draws some conclusive remarks.

Part I BACKGROUND

Chapter 2
Reliability Characterization

Reliability is the aptitude of a product or system to perform as intended (i.e., without failure) within its specified performance limits, for a specified time and in its life-cycle environment. Reliability characterization refers in general to all methods and procedures to measure how reliable a device is.

2.1 Reliability definition

The IEEE defines reliability as: "The ability of a system or component to perform its required functions under stated conditions for a specified period of time" [**2**].

Mathematically, reliability is the probability of a system to works correctly up to time *t*. One simple, commonly used model, is the exponential distribution. In this model the failure rate **λ** is a constant and the reliability R(t) at time t is given in eq. (2.1).

$$R(t) = e^{-\lambda t}, \lambda = \text{failure rate} \tag{2.1}$$

Reliability theory and reliability engineering make extensive use of the exponential distribution because of its *memoryless* property. This property is well-suited to model constant hazard rate components or systems. Memoryless means "the past has no bearing on the future behaviour". The probability that a component fails in the near future is always the same and doesn't depend on its current age. Every instant is like the beginning of a new random period, which has the same distribution regardless of how much time has already elapsed. Exponential distribution is also very convenient because it is easy to combine failure rates of independent components to find a reliability model of a complex system.

However, the exponential distribution is not appropriate to model the overall lifetime of organisms or technical devices, because their "failure rates" are not constant: more failures occur for very young and for very old systems.

The life cycle of a population of semiconductor devices, and of any living organism, can be graphically represented with a curve called "bathtub curve" (Figure 2.1), which models the cradle to grave instantaneous failure rate vs. time. While the origins of the curve are still uncertain; we can find studies from as far

as 1693 [**3**] showing human beings life expectancy having this kind of bathtub curve behaviour. The curve consists of three periods: infancy, useful life and wear-out. The earliest period, with steepest part of the curve, has the highest but decreasing failure rate, known as infant mortality,. Alternatively, the flat part of the curve, known as useful life (normal life) or random failure, depicts the lowest failure rate, relatively constant over an extended period of time, here is where the exponential distribution makes sense. The rightmost part of the graph, where the curve goes up again, represents the increasing failure rate when reaching the end of life, due to intrinsic material issues and accumulative electrical or mechanical stresses.

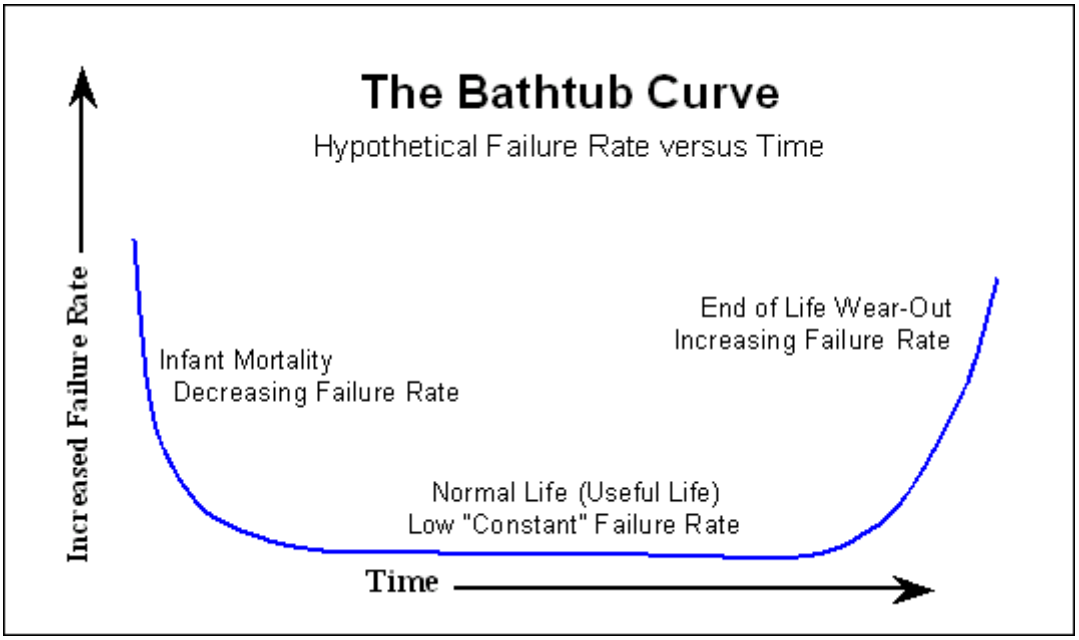

Figure 2.1 The bathtub curve

2.1.2 Associated standards

The International Electrotechnical Commission (IEC) develops and maintains international standards that provide systematic methods and tools for dependability assessment and management of equipment, services, and systems throughout their life cycles [**4**]. The commission defines dependability as “the collective term used to describe the availability performance and its influencing factors : reliability performance, maintainability performance and maintenance support performance” [**5**]. Consequently, in systems engineering, reliability is one of the performance characteristics contributing to asses a system's dependability.

Reliability engineering is related to safety engineering and system safety: they use similar methods for their analysis and may require mutual feedback. While reliability engineering focuses mainly on costs of failure; the focus of safety engineering is normally not on cost, but on preserving life and nature. High reliability levels are usually necessary to obtain high safety levels.

Functional safety ensures adequate protection against each significant hazard affecting any system, and is a concept applicable across all industry sectors. It is crucial to enable the use of complex technology for safety-related systems. It assures that the systems offers the necessary risk reduction required to achieve a proper safety level for an equipment. IEC 61508 [**6**] defines appropriate means for achieving functional safety in electrical/electronic/programmable electronic safety-related systems.

As the use of electronic components responsible for safety and mission-critical parts raises the necessity for high-dependability systems, different application fields have determined their own standards, defining and providing guidelines in the managing of components and systems reliability and safety.

In particular, car manufacturers are adopting the ISO 26262 standard [**7**], which is an adaptation of the IEC 61508 for Automotive electric and/or electronic (E/E) Systems in series production passenger cars. ISO 26262 addresses possible hazards caused by malfunctioning behaviour of E/E safety-related systems and their interactions. It demands a number of auditing processes during the whole product's lifecycle, to insure high reliability and mission safety throughout the system useful life. Remarkably, in mission testing for error detection demands the adoption of on-line self-test technique as an essential test process in critical E/E vehicle parts.

Another example of a domain-dependant standard dealing with Reliability, Availability, Maintainability and Safety (RAMS) issues is CENELEC EN pr50126 [**8**], (still under ratification process). In particular in its part 4 [**9**] gives guidelines for their specification and demonstration in electrical/electronic railway equipment and programmable electronic systems.

2.2 Characterization methods

Various methodologies aiming at assessing systems reliability have been the subject of scientific and industrial research. They are overall called *testing*, generally meaning measuring the output response to specific signals (usually called test patterns or test vectors) applied to the inputs, and comparing them with the known good response (golden solution) obtained by simulations. There exist different testing strategies, and they are also applied at different moments of a semiconductor device lifetime, in order to characterize its reliability.

Tests may be structural or functional. The first category includes all testing techniques that take advantage of deep knowledge of the internal structure of the Device Under Test (DUT). This kind of test checks that each element of the device is working as expected. The stimulus to be applied are simpler, because they target one element, so they can highly benefit from automation. However, this kind of test lacks overall view of the device and also it is difficult to assess a device's performance with this kind of techniques.

Tests belonging to the latter category do not need any information about the internal architecture, they use the functional specification of the device. They control that the DUT behaves as expected, and are useful to measure performances, though determining the proper stimuli to perform a functional test is usually a job requiring a lot of expertise that relies on the test engineer.

Another possible classification is according to the lifetime phase in which they are applied, roughly in two categories:

- Manufacturing testing
- On-line testing

Figure 2.2 shows an overview of the semiconductors characterization process, with all the testing steps a single digital device may suffer along his lifetime. In the next sections we will concisely describe the characteristics of each methodology, the equipments, tools and methods used, and the results that can be obtained.

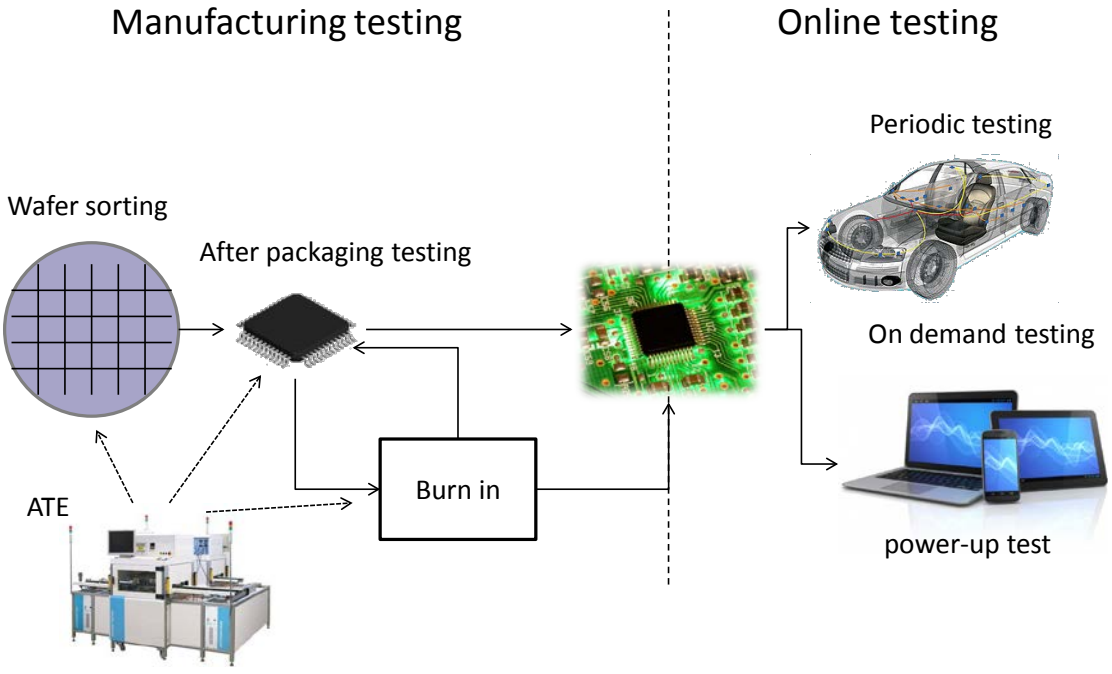

Figure 2.2 Semiconductors characterization process

2.2.1 Manufacturing testing

Manufacturing testing allows the evaluation of the performances of the target device at production time.

Semiconductor devices are very sensitive to impurities and particles. To manufacture these devices it is necessary to manage many processes while accurately controlling the level of impurities and particles. Consequently, even if the device's design was verified to be correct, there is no guarantee that the manufactured device is compliant with the design requirements. Therefore, manufacturing testing comes into role.

A methodology and equipments that can verify that each manufactured chip behaves as the reliable designed circuit and has no defects, at least immediately after manufacturing process. The manufacturing test main goal is to classify good from bad devices. Additionally, it can help in determining if there is any phase of the fabrication process that is systematically introducing a defect in the produced chips, by performing some kind of diagnosis.

In general, manufacturing test consists in applying a test pattern to a device under test by means of an external Automatic Test Equipment (ATE), sometimes profiting from a Design-for-Test (DfT) feature present on the devices itself, collecting the responses from the DUT and determining if the device is good or not. Eventually some extra data can be collected to perform a diagnosis of the found faults and aide in improving the fabrication and/or design process.

2.2.1.a Wafer Sort & After Packaging Final Test

Chip fabrication is a very complex process, performed in various steps. Tests are executed in various stages, in order to prevent expending money, time and efforts in finishing a non-working device.

When a wafer is ready, *wafer test* (a.k.a. *wafer sort*) is performed to all dies present on the wafer, looking for functional defects, by applying special test patterns to them. Faulty dies are marked, so only *known good dies* will be packaged; also repairing may be a possibility in next manufacturing steps; or eventually, if the number of faulty individual integrated circuits is greater than a certain threshold, the whole wafer may be discarded.

Once the *known good dies* have been packaged, a final test is needed to verify the packaging process itself didn't affect the devices, and also that the pin connections were correctly wired.

2.2.1.b Burn-in

In general, manufacturers want to avoid introducing to the market chips that will fail in an early stage (left-most part of the bathtub curve). If the application is critical, the justification is obvious; but even for non-critical applications, early failing chips may degrade the company's image and this is never an objective. To detect these common cases known as infantile mortality, a special procedure, called *burn-in,* is applied to them. Burn-in consists in subjecting the devices to particularly stressing conditions, in general determined by extreme temperatures and supply voltage, for a specified period of time. These stressing conditions work as a time machine for the chips (unluckily, only fast forward option is available). After the burn-in period, the final test is performed again to the devices. In this way, chips that were subject to infant mortality, won't make it into the market.

In order to characterize the bathtub curve, the complete burn-in process, that is covering all three stages of a device lifecycle, is to be performed on a statistical sample of products.

2.2.1.c Automatic Test Equipment

For all the manufacturing tests previously described an apparatus is in charge of applying the tests to the DUTs. Using automation, the Automatic Test Equipment (ATE), is able to perform the testing process. Basic component of the system are:

A *computer* in charge of controlling the process and the different instruments that will be connected to the DUT.

A variable number of *instruments*, which performed the desired measures, applying stimulus and collecting results.

A *fixture* that is the physical place holder for the DUT, where it connects to the ATE.

Eventually, a *handler* to place the packaged chips in the fixture; or *probes* that connect directly to the DUT when wafer testing.

It can be from as simple as computer controlling a multimeter, to a complex equipment, performing many different analogue and digital measurements.

2.2.1.d Design for Test

With more than 1 billion transistors in a 22 nm technology microprocessor, and Moore's law [**10**] yet to be denied, the complexity of the devices under test is correctly expected to continue to increase. Considering this, the minimum number of test vectors is also increasingly impossible [**11**], even with the most

fast and efficient ATE. In a Very Large Scale Integration (VLSI) integrated circuit (IC) it may take thousands of years to fully excite all possible states of a DUT. Of course, nobody can afford such a long process. Moreover, due to complexity, not all possible states may be reachable just manipulating the primary inputs of a device.

This testing problem can be avoided and its solution is called Design for Test (DfT). Circuits are designed in a way that make them efficiently testable; in particular using the sequential parts of them. This design modifications help the circuits to be tested with an acceptable fault coverage and in an acceptable time and, furthermore, to overcome the problem of test access.

The added special features can allow the control and observation of deeply embedded nodes to verify circuit functionality and detect fabrication defects. Performance loss due to their inclusion must be minimal in normal (non-test) operating mode. DfT may also help in making the process of application and observation of test vectors particularly suitable to automation.

The most popular DfT techniques are structured scan-based approaches (e.g., scan chains [**12**], boundary scan [**13**]), but there are also other useful ad-hoc techniques to be applied at the design stage (e.g., test point insertion [**14**], partitioning [**15**]).

2.2.1.e Test pattern generation

We already stated that in order to test a device a set of test patterns are applied to it, usually by means of an ATE. However, a very time consuming and important part of the characterization process is the generation of these test patterns. Essentially, test patterns are the stimuli to be applied at the inputs of a device in order to obtain an expected output. If we want to thoroughly test a component, the stimuli should be chosen carefully, extended enough so all the device's parts are excited; but also minimum, so as to minimize the test application time and the memory needed to store the patterns.

Often, generation effort and time of the patterns are an issue in themselves; Automatic Test Pattern Generation (ATPG) profits from the Design for Testability features included in-chip in order to generate at minimum cost a set of complete and efficient patterns. This kind of patterns and their possible optimization (compaction, compression, etc.) suit well for scan-chain based testing.

However, depending on the DUT or on the type of test, scan-chains may not be a possibility, or at least not an optimal one. For example, mixed-signal devices or on line testing of digital devices can hardly benefit from scan-chains. In these cases, carefully crafted input test patterns may be manually designed by the test

engineer in order to properly excite the circuit under test; being this a usually onerous task.

2.2.1.f Test Results

Once the patterns are applied, the responses from the DUT shall be compared on-the-fly with the golden model; i.e., the expected good response; or collected to be compared in a latter phase. In order to minimize data to be processed (and/or stored), different compression schemes were identified. Usually, one or more signatures will be generated from all the answers the DUT gave to all the applied patterns, and it is only these signatures that will be compared with the expected ones. The go/nogo verdict for the device will be decided basing on this. These comparisons can be made off chip by the ATE applying the stimuli; or eventually the outputs from the signature analyzer can be checked by a built-in checking circuit.

2.2.1.g Diagnosis

Chip diagnosis is usually a process performed once testing is over, it usually implies the collection of more than the go/nogo response from test phase. This data will be processes, more or less automatically, in order to provide the diagnosis engineer with as much information as possible, so he/she can individualize a possible problem in the production chain.

2.2.1.h Built-In Self-Test

A further step in aiding manufacturing testing is generating patterns and evaluating results on-chip, this approach is called Built-In Self-Test [**16**]. This technique requires some dedicated logic to have pattern generation, response analyzer and test application controller implemented in the hardware itself [**17**].

The generation can be from just a ROM storing the patterns to a Linear-Feedback Shift Register (LFSR) generating pseudorandom patterns on the fly, and also different implementations of counters.

A modified Linear-Feedback Shift Register (LFSR), called Multiple-Input Shift Register (MISR) [**18**], is a well-known technique widely used to implement the signature analyzer.

Besides, some methodologies were proposed so that the BIST added to the chip can also be used, complete or in part, during online testing [**19**] [**20**].

2.2.2 Online testing

Online testing are a group of techniques aimed at detecting the occurrence of a fault during the product mission time and to correct possible misbehaviours.

Reliability of semiconductor devices may also depend on assembly, use, and environmental conditions. Stress factors affecting device reliability include gas, dust, contamination, voltage, current density, temperature, humidity, mechanical stress, vibration, shock, radiation, pressure, and intensity of magnetic and electrical fields. If we want to guarantee a system or component reliability throughout time, the manufacturing test is not enough. Other tests need to be performed during its useful lifetime. There are different methodologies for online testing with different levels of confidence and consequences for the application.

Start-up testing, when a system is booting, before putting it to actual service, testing of some or all of its components can be performed. This implies no consequence for the application, as it is not yet running, it is useful for systems that are consistently restarted throughout their mission time, like the Power-On Self-Test (POST) of a computer memory.

Non-concurrent online testing is another strategy consisting in testing the system while the application is running, but with some constrains. This is, performing the test in one component of the system or in one part of the component that is not being used for the main application while tested. For example, testing one peripheral that is not being used; or testing a part of the memory chip after properly saving its content to a region not under test; and restoring the content after performing the test. This strategy is less intrusive than the start-up test, in the sense that it allows the application to run, but anyway it somehow prevents the complete free behaviour of the application. This kind of testing can be applied at periodic intervals, or its execution can be triggered by a particular situation present at the DUT. Some safety standards required this kind of periodic testing for critical systems as a car electronic control unit.

The third strategy is concurrent online testing, it consists in testing the device or system while the application is fully executed in the device under test. This is, of course, the less intrusive strategy from the point of view of the user application. However, it is not easy to find the proper stimuli and results collecting and analysis methodology. One example of this kind of online testing is the concurrent error detection (CED) mechanisms, for memories and processors, where an output characteristics predictor and a checker, can detect errors while the application is running.

2.2.2.a Software-Based Self-Test

The principle of software-based self-test (SBST) is to run functional test patterns, based on the processor instruction set; i.e., exploiting processor resources to test the processor itself and the components around it [**21**].

SBST consists in forcing the embedded processor(s) to execute a carefully crafted test program, i.e., a sequence of instructions capable of thoroughly exciting possible device faults and propagating the fault effects to some observable point(s). The test program can either be stored in a non-volatile memory, or uploaded in a RAM immediately before the test execution. SBST does not require circuit modifications (therefore making it particularly suitable for the test of third-parties cores, which can hardly be modified) and may offer good defect coverage, since it is executed at the same speed of the normal applications. Moreover, it can be performed both at the end of the production process, and during the operational phase (e.g., for periodical on-line testing); when the functional approach is used for on-line test, the results are typically checked by the system itself.

This strategies is firmly included in the manufacturing flow of microprocessors. Industrial experiences, such as [**22**] and [**23**] have confirmed the suitability of the methodology. In [**24**] an interesting case targeting a multi-core processor is presented. In that experience functional patterns are loaded in cache and applied to each of the 8 Processing Units belonging to a 4 GHz multi-core server in order to perform partial-good device binning.

SBST is also an appealing alternative for identifying faults during normal operation of the product, by performing on-line testing [**25**]. Several reasons push this choice: SBST does not require external test equipment, it has very little intrusiveness into system design and it minimizes power consumption and hardware cost with respect to other on-line techniques based on circuit redundancy. It also allows at-speed testing, a feature required to deal with some defects prompted by deep submicron technology advent.

However, some additional aspects have to be taken into account when dealing with test program generation for on-line purposes. The test program must first be able to properly excite the considered processor modules, and then, once the results have been produced, it must turn them observable from the outside in a transparent way that does not affect the normal operation of the mission application. The most important constraints for on-line testing include:

- *Preserving the processor status*: the status of the interrupted mission (i.e., the processor status register content) has to be saved and restored at the end of the test.

- *Execution time*: duration must be as short as possible complying with the requirements stated in the adopted safety standard [**7**].
- *Memory content*: it is crucial to prevent mission software and test programs from overriding information belonging to other processes. Code and data memory belonging to the test procedures must be clearly defined and limited considering the system memory map of the device.
    - *Code Memory footprint*: the code memory space required by the test programs should not be excessive, and it must conform to the established memory limits.
    - *Data Memory footprint*: the data memory space must also be as short as possible. However, the data memory placement can play a significant role with respect to the effectiveness of the generation process.

In the next chapter particularities and constraints in the testing of different electronic devices will be detailed.

Chapter 3
Characterization - Different devices, different strategies

3.1 Systems-on-a-Chip

With the advent of SoC, the low-cost concept has become a common denominator among test generation and test application.

In fact, in the SoC terminology, the term low cost is commonly used to classify a set of strategies and equipment that exploit DfT features included on a chip to reduce test costs without impacting its effectiveness [**26**][**27**][**28**][**29**].

The costs of SoC test procedure involve many factors, that are primarily test pin count, application frequency and tester memory depth for pattern and data storage. Many efforts have been done to blow it down in recent years [**30**].

Low-cost scan-based test approaches rely on design techniques that allow the minimization of the number of tester channels [**31**] and the tester frequency requirements [**32**]. In addition to (or in substitution of) traditional scan cells, these techniques adopt suitable DfT features such as decoders and phase-locked loop (PLL)-based circuitries; the former addresses pin count minimization, and the latter permits moving deterministic patterns in the chip at reduced speed, thus applying them at higher frequency. The introduction of test access protocols, as [**33**] and [**13**], to transport information inside the SoC architecture mainly addresses pin count reduction, often at the expense of the bandwidth [**34**]. Conversely, self-test procedures address frequency requirement mitigation, since it normally exploits internal or independent clock supply resources that do not request any external intervention.

Low-cost self-test approaches may be based on infrastructure intellectual property (I-IP) [**35**] or may employ functional parts of the DUT itself. The key point is that, once launched, a self-test procedure is autonomously applied until it ends. A further sub-classification for low-cost self-test approaches may include software-based self-test (SBST) [**36**] and BIST strategies [**37**][**38**]. Test procedures exploiting both SBST and BIST principles consist of at least three

parts: 1) a preliminary initialization phase aimed at loading at low-frequency the test microcode and/or setting parameters; 2) a self-test execution at high frequency; and 3) result download at low frequency.

Another strategy aimed at saving test costs is pattern compression. Such technique is intended to reduce the overall size of the test vectors to be applied to the DUT, thus reducing the test time. In the data compression techniques presented in [**39**] and [**40**], the process is performed by suitable software tools and consists in encoding the test vectors using as few bits as possible. Compressed data are then reconstructed, or decompressed, by ad hoc hardware decoders/decompressors placed on a chip or on the tester. In the specific field of embedded memory diagnosis, some approaches have been proposed to compress the results of the memory algorithm, finally providing a signature that synthesizes the amount of diagnostic data [**41**]; the drawback of this strategy is a potential loss in the diagnostic resolution.

Testing costs are stigmatized when the objective is diagnosis. In the case of volume diagnosis, the low cost is a must. Not only an increased quantity of information has to be downloaded but also diagnosis time may explode because diagnostic procedures frequently encompass many test iterations.

3.1.1 Microprocessors

Testing embedded microprocessors is one of the most challenging tasks to be performed at the end of the SoC production cycle [**42**]. Exhaustive testing is hardly a solution, since executing all the possible instructions, in all their feasible addressing modes, with all the potential data combinations, in all possible orders, starting from all the reachable initial states, may take extremely long testing times, and no one can afford waiting that long. Conventionally, testing strategies are largely based on the introduction of additional Design for Testability (DfT) hardware devoted to perform structural testing; scan chains [**12**] and Built-In Self-Test (BIST) [**43**] are well known and very popular solutions. Functional methods, such as Software-Based Self-Test (SBST) [**21**], are today increasingly used. Actually, hardware- and software-based techniques appear to be complementary and are often exploited in a more general testing plan including both of them [**44**].

There are various aspects to be taken into account when implementing a testing strategy for microprocessors: fault coverage, ability of the test to be executed at maximum speed, acceptable test time, guarantee of independence and Intellectual Property of the different cores to be tested in the system, area overhead introduced by the DfT module.

Moreover, when designing an on-line test strategy other issues must likewise be considered: preservation of the previous microprocessor state, ability to provide the diagnostic information and the fulfilling of the timing constraints considering that generally the time slice assigned to test execution is shorter than the test program itself, among others.

Today, SBST and other microprocessor functional testing approaches are gaining again popularity after many years in which scan-based approaches have been largely preferred. This is due to many reasons: the latest technologies show timing-related faulty behaviours that can be investigated in a more accurate manner using SBST [**45**], while avoiding over-testing and over-consumption; SBST techniques are cost and time effective, since they request few tester channels and limited memory amount on the tester; the self-test program can be stored in a non-volatile memory and activated also during the component lifetime to perform on-line testing. In the automotive and other safety-critical fields, emerging standards [**7**] and regulations mandate high fault coverage figures and usually require in-field testing (e.g., periodic on-line testing, power-up testing) that can be more easily implemented through SBST.

Many efforts have been invested in the past 30 years on the functional and SBST topic. Academy [**46**] [**47**] [**36**] and industry [**48**][**49**] have proposed many techniques solving general test problems and giving solutions to functionally reach the highest possible fault coverage.

Current state-of-the-art techniques include different strategies able to generate test programs resorting to manual and automatic approaches; generally, the proposed methodologies are generic enough to be easily adapted to various processors. New grading techniques to rapidly characterize test programs have also been proposed [**50**]. However, reducing costs related to both test program fault grading and test application is still an open issue.

Some interesting approaches tried to merge SBST principles with smart architectural solutions. In [**22**] a solution was proposed to generate suitable instruction sequences for testing, which are stored in the processor cache for sake of efficiency. In [**51**] a hybrid SBST technique was introduced that merges pseudo random pattern generation and functional processor behaviour knowledge.

Recently, the some specific issues raised by the on-line test of specific processor modules have been discussed and partly solved in [**52**].

Today, the problem of test is especially critical in the case of embedded processor cores. Their wide diffusion is increasing the challenges in the test arena. Modern designs include complex architectures that further increase test

complexity, as pipelined and superscalar designs. Single sub-components in a microprocessor may be autonomously tested by accessing their inputs and outputs through specific test buses built in the chip, and by applying specific test patterns, or resorting to integrated hardware, such as with Logic Built-In Self-Test (LBIST) [**53**]. Within this framework, a critical issue is the system integration test, where the whole processor and the interconnection between different modules have to be checked. At this level, one suitable possibility is to let the processor execute carefully crafted test programs.

3.1.2 Memories

3.1.2.a Memory SW BIST

We use the term SW BIST to denote a test solution targeting memories embedded in a SoC, based on performing the test through a suitable program executed by a processor inside the SoC: the program is in charge of executing the sequence of accesses to memory (for both read and write purpose) mandated by a given test algorithm (e.g., corresponding to a March algorithm).

Several published works, including those from industry (e.g., [**54**][**55**][**56**]), underline the fact that various memory defects existing in new technologies require the test accesses to be performed at the maximum speed (or at least at-speed) in order to be detected; this constraint is called Back-to-Back (or BtB) execution. The authors of [**57**] and [**58**] performed a careful analysis to understand how to match the BtB constraint when implementing test programs for CISC and RISC processors, respectively. Clearly, access to memory is performed through suitable instructions (i.e., LOAD and STORE in RISC architectures). However, implementing a March element in any assembly language requires addressing some critical issues:

- Loops
- Read result evaluation
- Memory address generation.

In the following we will detail each issue and summarize the limitations preventing the traditional SW BIST implementations to fully satisfy the BtB constraint.

Loops

When a March Element of a memory test is implemented in software, some instructions should be devoted to manage the loop in charge of repeating the required LOAD/STORE instructions for each memory word in the given order. Managing the loop requires introducing some index variable, which is updated and tested at each iteration; a conditional branch also needs to be executed.

During the execution of these instructions the memory is not accessed, thus , violating the Back-to-Back (BtB) constraint required for the detection of speed-related faults.

A possible solution relies on loop unrolling. However, this is a partial solution as we cannot infinitely unroll the loop in order to satisfy the BtB requirement for all memory access as mandated by a March Element. Moreover, even solving the problem partially leads to a significant increase in the code size (requiring larger memory).

Read result evaluation

March algorithms include read and verify instructions that require the read value from memory to be compared with the expected one. Usually, the check operation is interposed between the memory access ones, thus preventing the BtB constraint to be satisfied.

Memory address generation

March algorithms typically require to access all words in the target block in a given order. When BIST is implemented in hardware, address generation represents a major component of the required circuitry (in some cases up to 30% of the BIST, as reported by Mentor Graphics and others [**59**][**60**][**61**]). When SW BIST is considered, updating the memory address from one memory access to the following may require additional instructions, hence violating the BtB requirement. This issue becomes even more critical when the memory physical addresses do not match the logical ones, due for example to scrambling, mirroring and folding. In these cases scanning the memory words in the desired order may require complex pieces of code to be added, in charge of computing the physical address for every memory access. In [**58**] it is shown that some of these scanning sequences (e.g., fast-row) cannot be implemented when the assembly language of specific RISC processors is considered. Moreover, implementing memory tests usually require the use of multiple Data Background (DB) values. Once more, preparing the right DB value for each instruction may require additional assembly instructions, thus violating the BtB requirement.

Finally, it should be noted that the adoption of the SW BIST requires storing the code implementing the test somewhere in memory; in this way, not only the total amount of required memory increases, but we also somehow perturb the memory footprint with respect to the one required by the "pure" application.

3.1.3 Mixed-Signal devices

3.1.3.a Analogue-to-Digital Converters (ADC)

The dynamic performances of an analogue-to-digital converter (ADC) reflect the noise and the distortion introduced by the ADC in its signal path [**62**][**63**]. They are typically measured by injecting a single-tone sine-wave signal at the input of the ADC and computing a Fast Fourier Transform (FFT) of the output [**62**][**63**]. In the context of a Built-In Self-Test (BIST) technique aimed at computing the dynamic performances on-chip [**64**], it is required to replace the computationally intensive FFT algorithm with an alternative algorithm that can be implemented efficiently using digital resources. A well-known algorithm for doing this is the sine-wave fitting algorithm [**65**][**66**][**67**]. A variant of this algorithm with reduced complexity and, thereby, more efficient digital implementation, is proposed in [**64**] for the case of a BIST technique for a stereo switch-capacitor (SC) ΣΔ ADC. Another variant of this algorithm that is even less complex is proposed in [**68**] and makes use of the COordinate Rotation DIgital Computer (CORDIC) algorithm [**69**].

3.2 Sensors

3.2.1.a Micro-Electromechanical Systems (MEMS) Inertial Sensors

Micromachined inertial sensors, commonly known as accelerometers and gyroscopes MEMS (Micro-ElectroMechanical Systems), are one of the most important types of silicon-based sensors; indeed they reached 25% of the MEMS market in 2008 according to market analysis reports [**70**] and their market is forecasted to continue to grow. In fact, MEMS inertial sensors are widely used in automotive and aeronautics applications and are becoming extremely popular (21% expected annual growth) in a wide range of consumer electronics products, encompassing smart phones, 3D game consoles, personal media players, aided navigation systems, and camcorders stabilization systems. High volume production coping with the huge market demand is a crucial issue for the industry. Of course, the cost of testing is a major one within the MEMS manufacturing process; therefore, relevant efforts are currently being spent to speed up the test process by achieving high parallelism rates at a low cost and without losing screening effectiveness [**71**].

MEMS come from the integration of mechanical elements, sensors, actuators and electronics on a common silicon substrate through micro-fabrication technology. Their complex nature makes the testing issues even more challenging than for conventional semiconductor integrated circuits; advanced CAD and mechanical tools are needed to enable testing of a MEMS device in all stages of its

production. Final test flow encompasses many actions to be performed at different realization steps, as it is graphically shown in Figure 3.1.

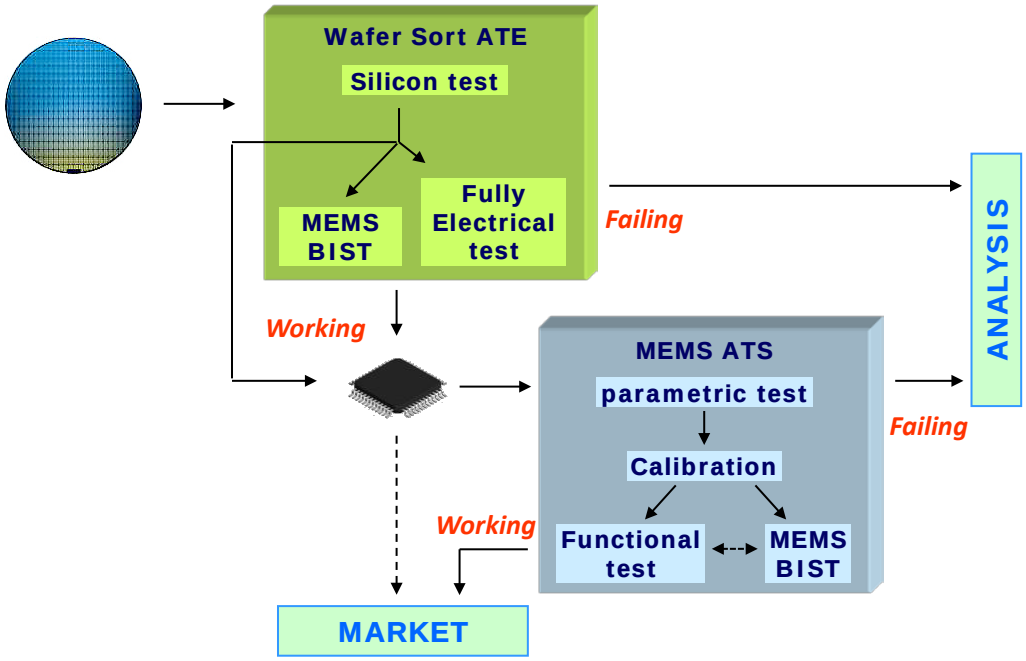

Figure 3.1 MEMS testing flow.

First, silicon parts of MEMS are tested at the wafer level by means of electrical wafer sort using an ATE (Automatic Test Equipment), possibly exploiting Design for Testability circuitries included on-chip, such as scan chains and self-test engines, similarly to common integrated circuits. This test phase only partially proves MEMS goodness; devices passing this preliminary stage are packaged and have to undergo a second test step that aims at checking their overall functional behaviour. Since MEMS are designed to respond to physical stimulation (i.e., movements, hits, sounds) with electrical signals, such a testing phase requires the application of physical stimuli beyond the electrical ones. After the parametric testing, aimed to assure the conformance of the accelerometer to the mechanical and electrical requirements, two consecutive operations are performed: *Calibration* (or Trimming) and *Functional Testing*.

Calibration is compulsorily performed on every single MEMS sensor before proper usage. It suits to find out some inherent constants (or trimming values) related to the device working principle [**72**][**73**][**74**]; trimming values change from chip to chip and are stored inside the Device Under Test (DUT) in a proper manner, such as using trimmed resistors, fuse transistors, ad-hoc registers or some kind of non-volatile memory, like a flash memory. Following calibration, functional testing of a MEMS component consists in applying a known physical stimulation and reading the device's output; if the measured value differs from the expected one the component is rejected, otherwise it is shipped to market.

Detailed descriptions of infrastructures (often known as Automatic Test Stations, or ATS) and methodologies for inertial MEMS calibration and testing can be found for accelerometers in [**75**] and for gyroscopes in [**76**][**77**]. Such equipment may be capable of performing the calibration and test process for many devices in parallel.

Many commercial MEMS implement self-test techniques [**78**][**79**][**80**][**81**][**82**], often based on BIST modules able to test the mechanical system through electrical stimuli. However, typically none of the preceding MEMS BIST techniques can be used to replace the traditional manufacturing test. The reason comes from the fact that considering the fabrication variations, the electrostatic force has to be calibrated first for each individual MEMS device before the device operates in self-test mode, and the calibration process requires the device to be thoroughly tested using external test equipment [**83**]. No matter the fact that BIST approaches are widely used for in-field, offline or online testing, they cannot perform sensor calibration if the BIST itself isn't calibrated firstly. Thus, calibration needs to be performed on each device by means of an ATS at some point in time before releasing it to the market.

Some efforts have also been done to devise a fully electrical method to estimate the sensitivity of capacitive MEMS accelerometers in batch fabrication without the need of mechanical test equipment [**84**]. This approach proved to be useful in reducing the dispersion of the sensors sensitivity, with an improvement that could be enough, typically for low-cost accelerometers. However, functional test, needing mechanical test equipment, is still required at least on a sample batch of sensors, for each sensor model that is being calibrated; finally, it is important to note that functional test cannot exploit parallelism, since the tester has to calculate the trimming values for each DUT.

The second part of the thesis describes the contributions made to the state of the art in three different testing areas: SBST, BIST and ATE.

Part II
CONTRIBUTION TO THE STATE-OF-THE-ART

Chapter 4
Proposed test programs for SBST of microprocessors

A widespread strategy to perform online testing of microprocessors is Software-Based Self-Test (SBST). The next sections describe SBST programs generation strategies for three very critical units of pipelined microprocessors: Prediction units, Register Forwarding & Pipeline Interlocking units and Address Adder units. In particular for the Address Adder unit the online constrains are carefully taken into account and a suitable strategy is proposed.

4.1 Prediction Units

A Prediction Unit is a mechanism to support speculative execution in order to overcome the performance penalty caused by branch instructions in pipelined microprocessors. Being an intrinsically fault tolerant unit, it is hard to achieve a good fault coverage resorting to plain functional testing methods. In this chapter we analyze the causes for low functional testability and propose some techniques able to effectively face these issues. In particular, we describe a strategy to perform SBST on an specific type of prediction units: fully associative Branch Target Buffer (BTB) units. The unit's general structure is analyzed, a suitable test program is proposed and the strategy to observe the test responses is explained. Feasibility and effectiveness of the proposed approach are shown on a MIPS-like processor.

Speculative execution has long been used as a way to overcome performance penalties derived from branches present in the instruction stream of pipelined microprocessors. A typical pipelined architecture has at least Fetch, Decode and Execute stages; one instruction per clock cycle is fetched, but it is not identified as a branch instruction until it reaches the *DECODE stage*; moreover, the branch is normally not resolved (i.e., the computation to know whether the branch should be taken or not-taken and to get the target address of the branch are not performed) until the Execution step. So, once a branch instruction is fetched, in order not to stall the pipeline, the microprocessor must fetch the next instruction, with no deterministic information if it is the correct one. If the wrong next instruction is fetched, some clock cycles, depending on the processor branch delay slot, are lost since the processor pipeline is flushed and new instructions

are fetched instead. Here is where branch prediction comes into play. Many strategies have been proposed to tackle this problem; from static always taken or always not-taken approaches to more complex dynamics ones, such as the Branch Target Buffer (BTB) mechanism, based on statistical principles.

Prediction accuracy of BTB mechanisms varies according to the different prediction algorithms, reaching up to 98.1% [**85**], when applying combined approaches.

All branch prediction execution strategies are error resilient by nature, because ultimately, in the Execution step, the correct next instruction is known for sure. If an incorrect decision was predicted, the pipelined will be flushed and the processor will fetch the correct stream of instructions. This behaviour does not lead to any functional error but at most to a performance penalty of two or more clock cycles, depending on the architecture. However, the faults present in the prediction unit may have a negative impact on the processor performance.

In order to detect these faults, a suitable test strategy is needed. The use of structural approaches, such as scan test or BIST, is not always possible; moreover, some of them are not suitable for at-speed testing, require an expensive external tester, or may not be adopted by designers because they put at risk the IP protection; finally, they may not represent the optimal solution (they have some area overhead, may lead to extra power consumption and often do over-testing). For these reasons, sometimes functional approaches, like Software-based Self-Test (SBST), are preferred [**21**]. SBST consists in uploading to the processor memory a sequence of assembly instructions, the so called test program, and then forcing the embedded processor to execute them. The test program should be capable of thoroughly exciting possible device faults and propagating the fault effects to some observable points. This technique does not request any circuit modification, therefore making it suitable for the test of third-parties cores. Moreover, this kind of test is, by definition, performed at-speed. Finally, test programs generated following the SBST approach are well-suited to be used for on-line testing.

As mentioned, prediction units are intrinsically fault tolerant circuits; this means that faults may cause a performance penalty, but the faulty processor still produces correct results. In [**86**] a case study was shown where all the stuck-at faults in the branch predictor were performance degrading faults, without causing any functional errors. This particular issue makes Branch prediction Units (BPUs) hard to test by functional means. In [**87**] a SBST methodology to deal with this kind of faults in BTB-based prediction units accessed via a hash function was proposed: the method resorts to performance monitoring hardware in order to observe the test program responses; the fault coverage

reaches 97%, taking about 20,000 clock cycles. A pure functional test for Branch History Table (BHT)-based branch prediction units is presented in [**88**]: the approach exploits processor instructions to implement an algorithm largely used in memory testing to fully test the decoding logic of a memory; the algorithm obtains a 100% fault coverage and is suitable for different implementations of the considered unit.

In this chapter, we propose a new SBST strategy for small, associatively-accessed BTB-based speculative execution modules. This kind of prediction units are generally used in processors designed for deeply embedded control applications that use the BTB to accelerate the execution of loops, such as the Freescale e200 family of PowerPC™ cores (z0, z1, z3, z6) [**89**]. They are purposely small to reduce processor cost and power consumption. In the following, the prediction unit mechanism is described and analyzed, a template to develop the test program is provided and parametric formulas to calculate both testing time and test program size are presented. Finally, different strategies able to observe the branch prediction unit faulty behaviour are discussed, highlighting their main characteristics.

4.1.1 Branch Target Buffer-based Prediction Unit Behaviour

The branch target buffer is a small memory associated with the pipeline of a microprocessor. It is used to predict branches result (taken or not-taken) and their target address. The BTB has many entries, each one with three fields: the branch address, its corresponding target address and some historic information bits (predictors) used to forecast the branch result.

The BTB works as follows: in the *FETCH stage* the current instruction address is searched among the instruction addresses present in the BTB. If there is a match, then a prediction is done using the predictors. If the prediction is taken, then the target address field is used to fetch the next instruction in the stream. In the *EXE stage*, where branches are resolved, the BTB is accessed again to verify if the prediction was correct or to create a new entry if the branch was not already present in the BTB. The predictors are updated, reflecting if the branch was taken or not-taken: if the prediction was wrong, the target address in the BTB is updated with the actually used target address and the pipelined is cleared. Figure 4.1 shows a schema of the Prediction Unit and its interaction with the pipeline.

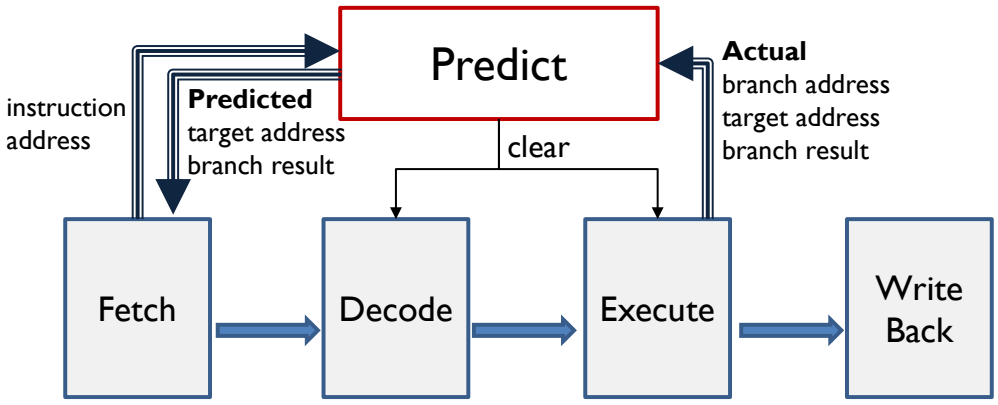

Figure 4.1 The prediction Unit and its interaction with the processor pipeline.

A decision to be made when designing a BTB-based prediction unit is which prediction algorithm to use. This defines not only how the predictors are updated each time a branch instruction is executed, but also the way they are used to actually make a prediction; this means deciding, given a predictors value, whether the branch is predicted taken or not taken. An extensive study of different branch prediction strategies for BTBs is made in [**86**]. Algorithms based on branch history, with 1-bit or 2-bit predictors are the target of this work. In the 1-bit history, the branch result is predicted according to the result obtained the last time that branch was executed, only. In the case of 2-bit predictors, different strategies can be considered to define the transitions on the 4–states Finite State Machine (FSM), with quite similar results.

Figure 4.2 shows the FSM diagram of a saturated counter behaviour which is a frequent choice [**88**]. In the figure, the state name contains the prediction and the state code; the label of the transition is the actual result of the branch. The counter is incremented each time the branch is taken, decremented otherwise; the MSB of the counter is used to predict the branch result (0 meaning not-taken, 1 meaning taken); in this way the mechanism actually implements a majority voter of the last 2 executions of the branch. If the branch is not yet present in the buffer, usually a static prediction strategy is applied (e.g., always taken).

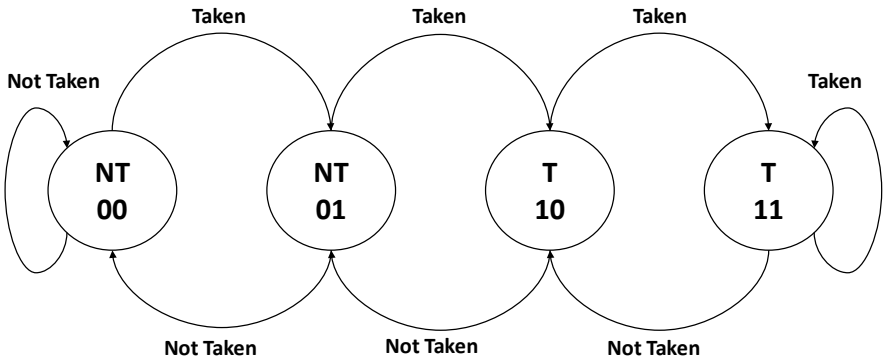

Figure 4.2 State diagram of the 2-bit saturated counter prediction algorithm.

Another issue to consider is how many entries the buffer shall have and how they will be accessed: fully associative, set associative, hash, etc. This work focuses on small buffers, mostly implemented with registers and with fully associative organization. Consequently, both in the *FETCH stage* (to read the prediction from the BTB) and in the *EXE stage* (to write and/or update the BTB), a comparison between the current instruction address with all the branch addresses already present in the table is performed. This kind of prediction units are purposely small to reduce cost and power and are especially expected to accelerate the execution of loops [**89**].

4.1.2 BTB-based Prediction Unit Architecture

Let us analyze the prediction unit architecture, dividing it in the three conceptual sub-blocks shown in Fig. 3:

- *Branch Target Buffer*: it contains the memory elements
- *Comparators*: they implement the fully associative mechanism.
- *Prediction & Control logic*: this block corresponds to the logic to provide the prediction and possibly update the BTB.

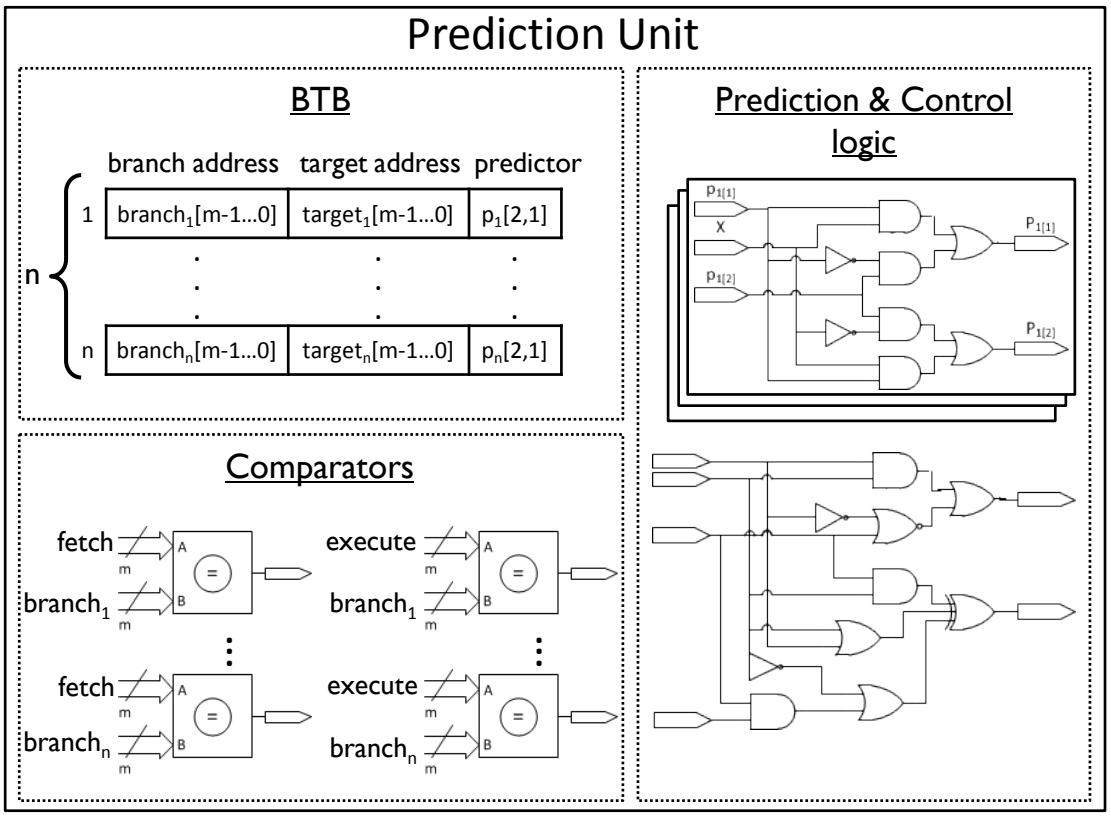

Figure 4.3 Prediction Unit schema with 3 sub-blocks.

The memory elements implementing the buffer occupy a relatively small area, but their particular characteristics and importance make them worth to be analyzed individually. Each of the n entries in the table has $m + m + p$ bits, where:

- n is the number of lines of the buffer and depends on the design, varying usually from 1 to 16 lines in this type of small, register-implemented BTBs
- p is the number of predictor bits used
- m is the number of varying bits of the address bus of the processor; as an example, in a 32 bits wide address bus with 4 bytes long instruction words, m is equal to 30. In case of variable length encoding, the shortest possible instruction length is considered.

The comparators, on the other hand, occupy about half the area of the prediction unit. Being a fully associative accessed memory, a comparator per entry is required; moreover, because the BTB is accessed in parallel in two different stages, it employs two times as many comparator circuits as the number of entries in the buffer. Each comparator has 2 m-bit wide inputs.

Finally, the logic implementing the prediction algorithm and managing the whole unit occupies the rest of the area. The prediction algorithm circuitry is used to

update the predictor bits in the BTB, and is usually replicated for each line. The control logic commands the filling of the BTB according to the comparators results and drives the outputs of the unit towards the pipeline.

4.1.3 Proposed methodology to test microprocessors' BTB

We propose a SBST algorithm for fully associative BTB architectures, though it may also be used in the case of a set-associative organization, applying it to a single set. The test routine is developed bearing in mind the three previously identified sub-blocks of the BTB-based Prediction Unit:

- Branch Target Buffer
- Comparators
- Prediction logic.

4.1.3.a Test of the Branch Target Buffer

To test the BTB it is required to write and read any pair of binary complementary data values (or backgrounds). Writing to the buffer is quite direct: every time a new branch instruction is executed, its instruction address and target address are written in the buffer (new meaning it was not already present in the table). To write complementary backgrounds on both instruction and target address fields, not only the destination of the branch, but also the position in the code memory where this instruction is located must be considered. On the other hand, reading the buffer and being able to observe the effects of faults in these fields is not as straightforward. The buffer is read on every clock cycle, in order to compare the address of the current instruction with the one present in the table. This comparison takes places both in the *FETCH stage* and in the *EXE stage*. In the latter one, if the instruction address in the table matches the actual instruction address in the *EXE stage*, a comparison between the stored and the used target address is also made. So, in order to observe the target address field in the buffer, each branch instruction should be executed twice: the first time it is written in the buffer, while in the second one it is used in the *FETCH stage* and verified in the *EXE stage*.

4.1.3.b Test of the comparators

To thoroughly test an m-bit wide comparator, independently on its low level implementation, it is stated in [**90**] that one can use a set of $2m + 2$ patterns: two of them are devoted to generate a mismatch in all the comparator bits, while a series of 2m patterns modify at every time only one of the comparator bits. This set of patterns may correspond to a walking 1, starting from the MSB of one of its inputs going all the way to the LSB of the other input, plus the two all 1s and all 0s patterns. Figure 4.4 shows the schema of a comparator, and the patterns to be

applied to test it obtaining 100% fault coverage. Anyhow, being combinational logic, the order in which these patterns are applied is irrelevant.

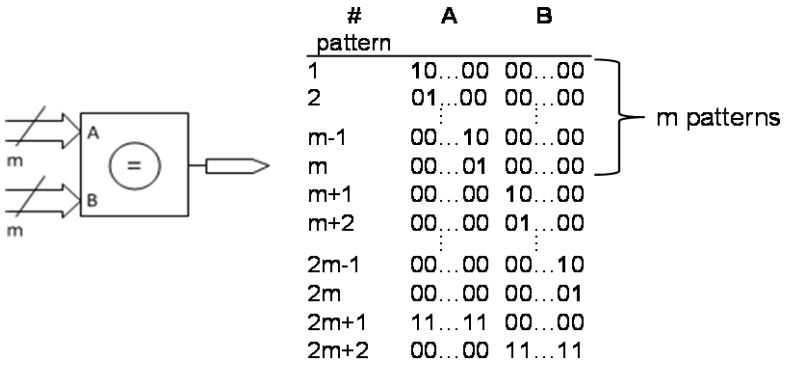

Figure 4.4 Comparator schema and patterns guaranteeing 100% FC.

In the case of the prediction unit, the inputs of the comparators are connected to the current address present in the stages participating in the prediction and to the branch address entries of the table. To exemplify, let us consider input B of Figure 4.4 to be the instruction address entry of a 1-line table ($n = 1$), and input A the actual address present in the *FETCH stage*. To test this comparator the all 0's pattern should be first written into the buffer: this is done inserting a jump instruction at the start address (i.e., 00…00) of the code memory. Then, a sequence of jump to start instructions have to be executed, placed in suitable addresses, so as to make the program counter (PC) evolve from pattern 1 to pattern 2m (Figure 4.4 column A). Finally, the last jump to start instruction of the sequence is placed in the last address of the memory, writing the all 1's pattern in the buffer. The jump instruction to the start address may be implemented with an indirect jump, using a register initialized to the first pattern and subsequently updated to go through all the aforementioned patterns. In this way, as the comparisons take place at each fetch, all of the 2m+2 required patterns are applied to the comparator. To illustrate this example we considered an address bus of 8 bits and 2-byte wide instruction words, meaning $m = 7$. A snippet of a list-like assembler pseudo code for the 1-line BTB of the example is shown in Figure 4.5, supposing r1 initialized with the value 00000010.

It is possible to note that the piece of code presented in Figure 4.5 does not apply the test patterns presented in Figure 4.4 in the same order. Clearly, it is not convenient to fix the BTB entry (input B) to 0x00 while applying the first patterns of the marching one sequence (m patterns) to the input A, since it requires the use of jump instructions that do not modify the state of the BTB entry when the program goes back to the start label. On the contrary, the proposed strategy interleaves the first (1 to m) and second ($m + 1$ to $2m$) set of

patterns presented in Figure 4.4, by using regular jump instructions *jmp* that modify the BTB.

```
org 0000
start:          jmp r1
(00000010)      mov r1, 00000110
(00000100)      jmp start
(00000110)      mov r1, 00001110
(00001000)      jmp start
(00001110)      mov r1, 00011110
(00010000)      jmp start
(00011110)      mov r1, 00111110
(00100000)      jmp start
(00111110)      mov r1, 01111110
(01000000)      jmp start
(01111110)      mov r1, 11111100
(10000000)      jmp start
(11111100)      mov r1, 00000010
(11111110)      jmp start
```

Figure 4.5 Test program for the comparator in a BTB with n=1 and m=7.

For larger tables, but which anyway have not enough entries to admit all the patterns, a jump to start address is to be added every n instructions, being n the number of lines of the buffer, whereas the n-1 branch instructions in the middle point to the next relevant address. Such a test program guarantees that the all 0's pattern is always present in the table to perform the necessary comparisons. Besides, the whole procedure should be repeated for each of the n lines of the buffer, assuring all patterns are applied to every comparator. This can be accomplished repeating the same cycle and taking care that each repetition starts storing the branch information in the subsequent line with respect to the previous cycle. If necessary, to align the entries in the table, padding jump instruction can be added at the end of the cycle.

Generally, comparisons take place during two pipeline stages: fetch and execute stages, but a comparison may also occur in the *DECODE stage* in some implementations. Nevertheless, with the proposed code, all the comparators related to one BTB row are excited concurrently.

4.1.3.c Test of the prediction logic

Finally, to test the prediction logic, conditional branches are used, in order to go through all the stages, exciting every transition of the FSM implementing the prediction algorithm. A 2 bits predictor leads to 4 states; different algorithms lead to different transitions between these 4 states, as stated in section II. In this work we consider the FSM shown in Figure 4.2, implementing the saturated counter, but the test program can be adapted according to the algorithm implemented by the prediction unit under test with a limited effort.

With any adopted 2-bit strategy, one FSM per buffer line is implemented, each of which composed of four states. In order to guarantee the FSM to be in a certain state, three jumps are needed. Then, for each state, two possible next states are available. Finally, the instructions that excite the inputs of the prediction unit FSMs and make them change state are jump instructions. So, to exhaustively test all the FSMs in the prediction logic, at most $4 \cdot 3 \cdot 2 \cdot n$ jump instructions are needed.

In the 2-bit saturated counter approach, a strategy similar to the one proposed in [**88**] may be used, where 3 taken branches take the FSM to the initial state; then, 4 not-taken and 4 taken branches guarantee all possible transitions to be executed. This strategy minimizes the number of jump instructions to be executed for each counter to reach the 11 state; therefore, to test the whole prediction logic $11 \cdot n$ jump instructions are required. The code snippet in Figure 4.6 shows a possible implementation of a test program for the saturated counters prediction logic using the strategy proposed in [**88**]; *bgez* means *branch-if-great-or-equal-zero*. It is to be highlighted that this procedure is independent from the instruction or target addresses, from the size of the comparators or the number of entries of the buffer and it needs to be executed once for each line of the BTB.

```
sat-count:      mov   r2, 2
                mov   r3, 4
                mov   r4, 1
logic:          bgez  r2, taken
not-taken:      bez   r4, end
                dec   r3
                bgez  r3, logic
                mov   r2, 3
                mov   r4, 0
                jmp   logic
taken:          dec   r2
                jmp   logic
end:            call  initialize
```

Figure 4.6 Pseudo code of the test for the saturated counter prediction logic.

4.1.4 Test program for the BTB prediction unit

From the previous analysis a general test program is deduced, which combines the testing strategies for each of the three Prediction Unit sub-blocks in a unique SBST–ready program. The program is parametric with respect to the number of lines in the buffer (n) and the number of bits of the comparators (m).

The structure of the program is the following:

- The program is mainly based on a cycle of as many branches as entries in the BTB minus one; at the end of this basic cycle a branch to the start address exists, so as to excite the comparators with the previously described patterns.
- The basic cycle is repeated, modifying the target addresses, until all the patterns have been applied, i.e., the target address of the branches must vary from `10…00` to `00…01`.
- At this point a call to the subroutine that performs the conditional branches, detailed in Figure 4.6, is inserted.
- The whole procedure is repeated as many times as rows in the buffer. In this way each entry of the table is written with every pattern and compared to the all 0's value, and also the all 0's value is written to each entry and compared to every pattern. The prediction logic associated to each entry in the BTB is also tested.

The described main program is associated with proper initialization and update routines. A schema of the proposed test program is shown in

Figure 4.7.

```
org 0000
start:            jmp r1
org (pattern 1)
                  jmp r2
org (pattern 2)
                  jmp r3
                     .
                     .
                     .
org (pattern n-1)
                  call update
org (pattern n)
                  jmp r2
org (pattern n+1)
                  jmp r3
                     .
                     .
                     .
org (pattern 2*( n-1))
                  call update
                     .
                     .
                     .
org (pattern m)
                  call sat-count
```

```
org entry point

initialize:
            mov r1, pattern 1
            mov r2, pattern 2
            .
            .
            .
            mov rl-1, pattern n-
      1
            jmp start
```

```
update:
            sll r1,(n-1)
            sll r2, (n-1)
            .
            .
            .
            sll rl-1, (n-1)
            jmp start
```

Figure 4.7 General test program schema.

It must be noticed that the length of the main test program grows linearly with the number of bits to be compared; in the fully associative case, this is equal to the number of varying bits of the address word. On the other hand, the size of the initialization and update routines increases linearly with the number of entries of the BTB.

4.1.4.a Testing time

The testing time is almost optimal, corresponding to one branch instruction and one update instruction for each pattern for each line, and the additional subroutine calls and branches to the start address are sufficient to test the BTB and the comparators. To calculate the testing time of the prediction logic, it should be noticed that correctly predicted branches take 1 clock cycle, while a wrongly predicted branch takes 2 or more clock cycles. This delay depends on the number of stages of the pipeline that are flushed when the wrong prediction is done, i.e., the time needed to clean the pipeline. Eq. (4.1) summarizes the concept, assuming all instructions require 1 clock cycle and f is the number of pipeline stages to be flushed when a misprediction occurs.

$$Test\ time = n * (2m + 9f + 36) + 2 * {}^{m}/_{n}\ [clock\ cycles] \quad (4.1)$$

4.1.4.b Observing the BTB behaviour

In order to observe any possible performance-degrading fault affecting the BTB, it is possible to exploit one of the following mechanisms:

- the performance counters [**91**] existing in many processors and able to monitor the number of correctly/incorrectly executed predictions,
- some timer able to measure the performance of the processor when executing a given piece of code, exploiting the fact that mispredictions imply a longer execution time,
- some ad hoc module added to the system and able to monitor the bus activity [**92**].

4.1.5 BTB SBST experimental results

The feasibility and effectiveness of the proposed approach have been evaluated on a miniMIPS processor [**93**] synthesized with an in-house developed library and resulting in a 57,776-gate circuit. The miniMIPS architecture is based on 32-bit buses and includes a 5-stages pipeline. An embedded timer was included into the final implementation making it possible to observe the branch prediction unit behaviour, as described in section 4.1.4.b .

The logic simulation was performed using ModelSim SE 6.4b by Mentor Graphics, while the circuit fault coverage was evaluated using TetraMax v. B-2008.09-SP3 by Synopsys.

The BTB implemented in the miniMIPS is fully associative, has 3 lines (n=3) and the address word is 32 bits wide, leading to m=30 bit wide comparators. The two least significant bits are not compared nor stored, since they are always 0 (instructions are 32 bit wide). Three stages are implied in the prediction mechanism: fetch, decode and execute (f=3). The Branch Prediction Unit counts 3 fully associative entries supported by 10 comparators (9 for the instruction address and 1 for the target address). The size of the synthesized version of the unit is 4,076 equivalent gates, leading to 19,622 stuck-at faults.

TABLE I shows the fault coverage results, dividing them in the three identified functional subunits of the BTB, and their respective weight in the total number of faults. It must be noted that with the proposed approach the coverage for the table and the comparator is always guaranteed, whereas for the prediction logic, modifications should be done according to the prediction algorithm used, as stated in section 4.1.3.c .

TABLE I PREDICTION UNIT FAULT COVERAGE

Unit	Faults	Fault Coverage
BTB	1,217	99.34 %
Comparators	7,299	98.94 %
Prediction Logic	11,106	96.90 %
Total Prediction Unit	19,622	97.81 %

After an accurate analysis of the final results, we noticed that the faults still not covered in the considered Branch Prediction Unit were mainly related to two factors. Firstly, there are some not covered faults that belong to the circuitry devoted to handle conditional branches: these faults were not thoroughly excited since we hardly use jump instructions that use an address contained in a register as the destination. Secondly, there are some untestable faults due to redundant logic generated during the synthesis process used in this implementation. Following the previous considerations, we decided to include an additional piece of code oriented to specifically excite conditional branches, raising the coverage of the Prediction Unit to 99%.

The final test program includes three different sections: an initialization program that configures the processor core, as well as the external timer used during the testing procedure; the actual testing procedure, derived as described in the previous sections, and a third program in charge of collecting and comparing the execution results. The whole test program requires about 800 clock cycles to execute and counts about 500 bytes. These are interestingly low figures, mainly due to the fact that the proposed approach only needs a few extra instructions in order to configure and read the external timer enabled to observe the actual time required to execute the complete test program.

4.1.6 Conclusions about the BTB prediction unit SBST

This chapter proposes a strategy for the test of small BTB-based prediction units in pipelined microprocessors. The architecture and behaviour of the unit were detailed, and the test program was described for the general case, and then customized for a 3-entry fully associative BTB implementation embedded in a MIPS-like processor with a 5-stages pipeline. Stuck-at fault coverage of about 98% was achieved for the prediction unit, with a very short test time and small code footprint.

The method is appropriate and very effective for small BTBs implementing fully associative or set associative strategies, using a large number of comparators. The major advantage of the method is that it can achieve high fault coverage

without introducing any change in the processor hardware, and without even requiring detailed information about the BTB implementation.

4.2 Address Calculation unit

This chapter proposes a method for the generation of SBST programs to test on-line the Address Calculation unit of embedded RISC processors, which is one of the most heavily impacted by the on-line constraints. The proposed strategy achieves high stuck-at fault coverage on both a MIPS-like processor and an industrial 32-bit pipelined processor; these two case studies show the effectiveness of the technique and the low effort.

Different approaches have been proposed to test, by means of SBST techniques, processor modules such as the Address Calculation unit. However, in most cases, the previously mentioned constraints for on-line testing were not explicitly considered during the test program development. In some cases, for instance, the test application time is not suitable for on-line testing [**94**]; in others, the code size becomes also an unconsidered constraint [**95**]. Finally, in some approaches (for example [**96**]) the Address Calculation module is not explicitly targeted, and no special considerations have been taken into account regarding the placement of the data and code memory blocks devoted to allocate the on-line testing program. The resulting programs are, in any case, a good starting point and can be transformed into on-line test programs, as detailed in [**97**], where enforced constraints are considered.

We concentrate on the on-line SBST of a very sensitive component in pipelined microprocessors: the Address Calculation unit. This part of a processor is used to calculate the addresses for load and store memory accesses, usually summing a base address to an offset. The Address Calculation module is intensively used along the mission time; therefore, its components are more likely to early degrade than others (e.g., ALU ones). A failure in this module can provoke sneaky device behaviours, such as accessing wrong data in a still legal but misplaced address range; the effect is a deviated but not disruptive application execution that may not be revealed by hardware approaches, such as watchdog timers and software-implemented redundancy.

In the on-line testing context, the generation of SBST programs targeting the Address Calculation module is really critical because of the constraints imposed by the coexistence of the test routines with the mission application. Ideally, the mission application should freely access data and code memory; therefore, the test routines are constrained to use a limited memory address range for writing signatures and reading known data.

The proposed test generation method provides guidelines for the selection of the on-line compliant address range reserved for test purposes and includes a test program synthesis flow based on atomic blocks, which is very effective in terms of stuck-at fault coverage and takes into consideration the on-line constraints.

4.2.1 Generation flow for on-line test programs

In this chapter we detail the constraints and a generation method to test the Address Calculation unit of a microprocessor during the mission phase of the device.

The aim of this chapter is to introduce an effective strategy to generate SBST programs, or routines, suitable to be run periodically during the device mission. The illustrated strategy falls into the non-concurrent on-line testing domain because the mission application is interrupted at regular intervals to let the self-test routines run.

The processor component specifically targeted by the proposed approach is the Address Calculation unit; this module is in charge of calculating the addresses for memory access operations performed when load and store instructions are executed. This computation is performed by a dedicated adder, whose purpose is to sum an offset to a base value to address the RAM memory for reading or writing a value.

Usually, this adder is not part of the processor ALU, and thus it does not perform any arithmetic computations required by instructions like add and sub. Testing an adder is often deemed as a trivial task, but in the case of the address generation module, controllability and observability are limited, and on-line requirements pose additional constraints.

The criticalities in testing this module by using a software-based approach are mainly due to the type of instructions (load and store and all their variants) that activate the Address Calculation. In fact, a test program including many of such instructions may potentially induce some undesirable effects:

- store instructions may corrupt the mission data and compromise the correct resuming of the system;
- load instructions may retrieve data from memory zones (i.e., the parts containing the mission application variables) whose content can hardly be forecasted a priori, therefore compromising the signature generation, no matter how it is calculated.

Taking these factors into consideration during on-line test generation is a must; careful planning of a SBST generation campaign should early consider memory access constraints.

Figure 4.8 shows the conceptual view of the generation approach we propose. The main block in the scheme is called *Generation flow*. Other than usual inputs, such as the netlist of the circuit and the fault list of the tackled module, the *Generation flow* considers a set of constraints imposed by the coexistence of the test routines and the mission application, and leverages on a SBST atomic program template whose structure is tailored to take the on-line constraints into account.

The *Generation flow* is composed of a set of steps that are detailed in the next paragraphs; along the generation process, the fault coverage is a feedback measure that allows the program generation to proceed towards a set of routines guaranteeing high fault coverage. The final result of the generation process is a test suite mainly performing a sequence of load and store instructions, along with some arithmetic instructions devised to setup base and offset values for memory access.

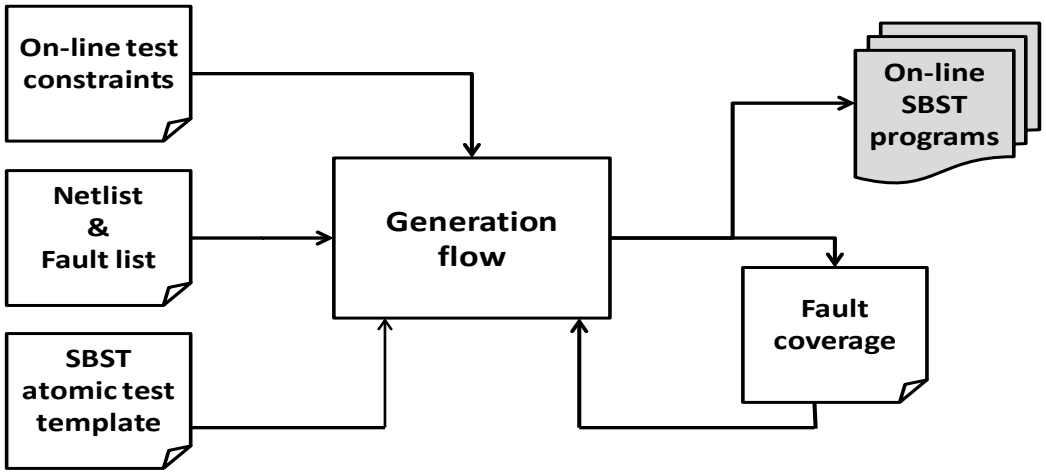

Figure 4.8 Conceptual view of the proposed generation approach.

4.2.1.a On-line test constraints

In the specific case of the Address Calculation unit, on line limitations are very strict:

- Store instructions have to write only in reserved memory locations, possibly contiguous in the addressable space
- Load instructions have to read from memory sections never varying in content, possibly reserved for this purpose
- Memory zones reserved for writing and reading should be separated, to avoid the aforementioned problems
- Both read and write zones need to be programmed with suitable initial values

- Arbitrary values are suitable for reading zones, provided that they are diverse within the zone; in this way the method minimizes the aliasing potentially stemming from a compromised memory access (e.g., when the reading zone is all 0s, accessing a wrong address in the range may cause non-detection)
- The all 0s value is fine for the writing zone, since values transferred to the memory are sourced from the reading zone, which is properly filled.

The setup of the initial memory content is a critical operation, since it may be affected by faults in the same logic for Address Calculation to be tested. To avoid undesired fault effects, direct memory access (DMA) driven initialization is suggested; for example, the reading zone may be filled by copying part of the test routine code which is invariant, while this is not guaranteed for the mission program code.

The first issue arising from these limitations concerns the selection of memory zones. This selection directly reflects in the effectiveness of the generation process. Ideally, reserving a single small memory portion is desirable; however, a too small address range may prevent the achievement of a high coverage. For making an effective selection, it is useful to define first the acceptable amount of memory *M* that can be reserved for testing purposes (called Test Memory). This portion of memory can be eventually shared with test programs to store signatures; therefore, we suggest that the test of the Address Calculation module should be the first module considered in a SBST suite development plan.

To select the range *R* defining the starting and ending addresses of the Test Memory given the dimension *M* of the available memory, the simple formula (4.2) can be used, where *N* is equal to the number of addressing bits; the formula assumes that the Test Memory is located in the middle of the address space of the whole memory:

$$R = \left[\frac{2N - 1 - M}{2}, \frac{2N - 1 + M}{2}\right] \tag{4.2}$$

The effectiveness of this choice is due to the fact that in this way the Hamming distance between the extreme address values is maximum, i.e., it is equal to *N*, allowing the Address Calculation adder outputs to exhibit a large value variance.

For instance, let us suppose that the addressing space is 32 bits, but only with 128KB available within the address range *R* data = [`0x40000000, 0x40001FFFF`] corresponding to an addressable subspace of 17 bits. Supposing to have 1KB to be allocated to test purposes, the selected address range is *R* = [216-512, 216+512-1] relative to the *R* data segment start address. Hexadecimal

values better show the effect of this choice considering the Hamming distance in the range R = [0x4000FE00, 0x400101FF]; all 17 less significant bits are changing values. Figure 4.9 visualizes the effect of such address range over the Address Calculation adder. As a counter-example, let us locate the 1KB test data on top of the memory addressable space, in the range R = [0x40000000, 0x400003FF]; in this case, only 10 bits in the address range can be varied.

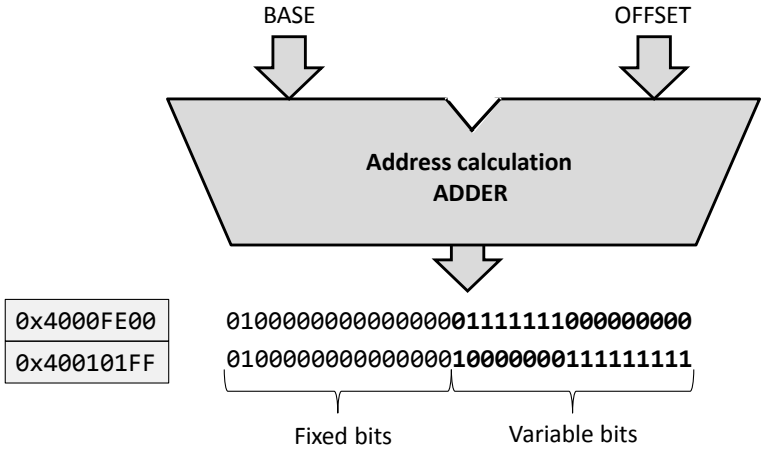

Figure 4.9 Effect of address range selection for Address Calculation adder test.

The identified memory space can be used either for writing or reading by defining two sub-zones.

As it can be noticed, the test effectiveness of the Address Calculation adder is strongly limited by the available address range, since several adder output bits may be fixed to a constant value. To overcome this problem, a viable solution can be based (whenever possible) on accessing other chip resources located out of the Test Memory (i.e., 0xBFFF000 may be a suitable complement to the previously selected address range). As an example, in most real cases the addresses reserved for peripheral core registers can be easily and safely accessed. Moreover, the selected complementary locations may be used for reading only, thus guaranteeing that their content is never changed.

The test program structure described in the next paragraph is suitably studied to cope with the identified requirements.

4.2.1.b Atomic block structure

Roughly speaking, test programs targeting processing modules need to apply suitable values to the module inputs by means of controlling instructions; some instructions are used to setup the data to be elaborated by the tested module

when a target instruction is executed, and then results are propagated to the processor outputs.

Testing the Address Calculation unit by using the SBST principles means reading and writing data from the memory at suitable locations. Let us consider the following generic memory access instructions (store and load):

```
sw rX, base (offset)

lw rX, base (offset)
```

When using such instructions, the actual address is calculated by adding the values provided as offset, and base, which usually correspond to two registers or to a register and an immediate value encoded in the instruction, respectively. The execution of this instruction excites the Address Calculation module by applying the base and offset values at the module inputs.

For testing sakes, base and offset have to hold suitable values to achieve fault coverage; in the on-line scenario, additional constraint must be considered and the resulting addresses must belong to a small memory range corresponding to the Test Memory.

To overcome the issue of performing effective sum operations while matching these strong constraints, we propose the usage of an atomic block devised to support the test generation process. The atomic block illustrated in the following can be used as a building element for the test program targeting the Address Calculation unit. The usage of similar blocks, called building blocks, was introduced in [**98**]; however, the authors did not consider the particularities of the Address Calculation modules of pipelined processors, neither the constraints regarding on-line testing.

In short, the proposed structure first loads a data value from a test memory location in the read reserved space; then this value is modified, and finally, it is saved again in a writable test memory zone. The pseudo-code of the atomic block is shown in Figure 4.10.

The first two instructions in Figure 4.10 (lines 1 and 2) load random values in the registers `rA` and `rB`. The value in `rA` is a constrained random address value selected within the readable Test Memory addressing range (rd constrained): a known value must be stored previously in memory at this address. On the other hand, the value placed in `rB` is purely random without any constraint. The addition and subtraction instructions of lines 3 and 4 prepare registers `rC` and `rD`, which activate the arithmetic adder; additionally, these instructions manipulates the registers involved in the load instruction placed at line 5, that

accesses the memory location at the address in rA, during which the Address Calculation module performs the addition between rB and rC. The memory value is read in register rE. The instruction at line 6 manipulates rE by applying the function f(rE, rD), which is aimed at merging the results of the memory read (line 5) and the arithmetic addition (line 4). During our experiments, the function f(rE, rD) was replaced by the logic XOR instruction on the registers rE and rD. The advantage introduced by this function is that both address calculation and arithmetic adder are tackled obtaining high levels of fault coverage without additional effort. The value in rE is later stored in memory completing the observability task of the atomic block for the test of both adders.

In the second part of the pseudo-code, (lines 7-9) new random values are loaded in rA and rB; both of them are constrained values, since they are used to calculate the destination address to store rE in line 9. Clearly, the addition of rA and rB must produce a value placed in the writable Test Memory (wr constrained). Finally, it can be noted that line 9 collects the information elaborated by the atomic block and uses a store instruction to send it to the appropriate location.

```
1.          mov   rA, RNDM (rd constrained) value
2.          mov   rB, RNDM (unconstrained) value
3.          sub   rC, rA, rB
4.          add   rD, rB, rC
5.          lw    rE, M[rB,rC]
6.          func  rE, rE, rD
7.          mov   rA, RNDM (wr constrained) value
8.          mov   rB, RNDM (wr constrained) value
9.          sw    rE, M[rA, rB]
```

Figure 4.10 Atomic block pseudo-code.

The structure proposed for the atomic block is as general as possible, so as to cope with different processor realizations. In the example above we supposed that the two elements involved in the memory access instructions correspond to a couple of registers. Clearly, depending on the targeted case, the adopted solution may include a couple of registers or an immediate value and a register.

4.2.1.c Test program building flow

A suitable test program can be built by exploiting the atomic block introduced in the previous section. Its final form is a sequence of atomic blocks including selected values for exciting and propagating the address calculation adder faults. Possibly, no loops should be included in the test program code since those constructs usually lead to a long execution time, even if they permit saving program code lines.

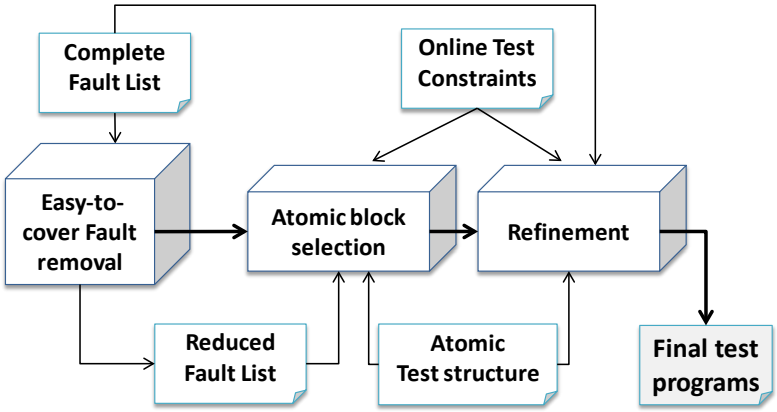

Figure 4.11 Proposed framework for on-line testing.

To perform accurate and fast selection of the required values, we propose a three-step generation process, whose graphical view is reported in Figure 4.11. The result of this flow is a program composed of a sequence of carefully selected atomic blocks to be sequentially executed. The three steps differ due to the set of faults they work on, and on the test generation method they adopt.

The proposed flow derives from this concept: in any circuit there are faults that are easy-to-cover, i.e., they are detected by a large set of patterns, while there are other faults that require very specific test sequences. Therefore, we propose:

1. to initially remove from the fault list a possibly large set of easy-to-cover faults, even using unconstrained test programs;
2. to produce focused atomic block values considering the remaining faults, in this phase the on-line constraints are taken into account; more in details
    a. when the required level of coverage is reached, the obtained test program is graded with respect to the whole fault list (including easy-to-cover faults previously removed)
    b. most of easy-to-cover faults will result covered also in this case; therefore
    c. the coverage figure will only slightly decrease;

3. to integrate some more atomic blocks with proper values to refine the test program.

The usual fault coverage trend observed along the generation process is shown in Figure 4.12.

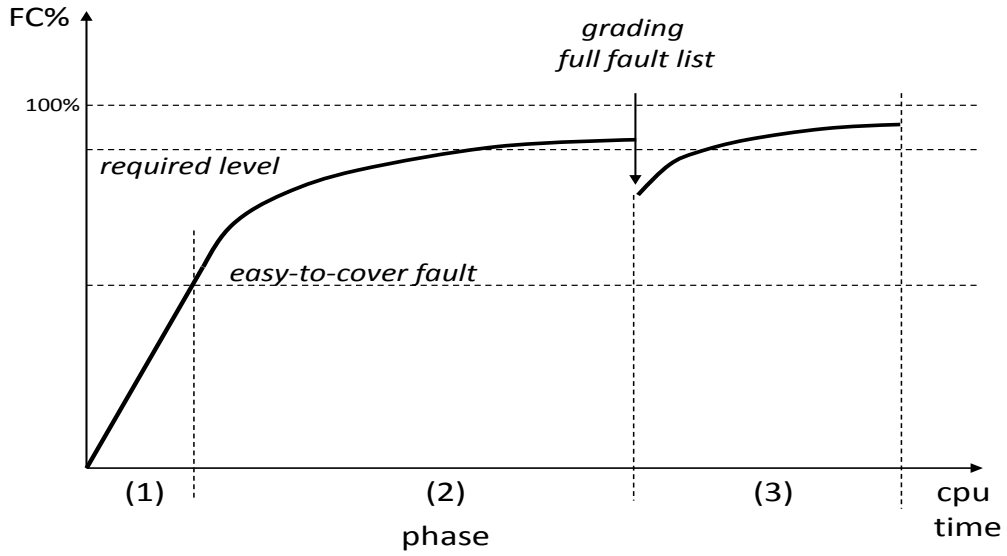

Figure 4.12 Fault coverage general trend along the generation process.

Figure 5. Fault coverage general trend along the generation process.

1. Easy-to-cover Faults Removal

In this phase a preliminary test program is written and evaluated without considering any on-line test constraint. This program may be composed of few manually selected instructions exciting the Address Calculation unit; particular cases, such as `0` values for base and offset, and random values may be included. As a result, many faults will be covered, which are said to be easy-to-cover. These faults are usually related to the logic parts of the address calculation adder located close to its inputs and outputs.

The coverage achieved by this process is usually quite high and up to 50% in some cases. The faults not covered are used as the input fault list for the next step.

2. Atomic block selection

Starting from the reduced fault list inherited from the first phase, the generation process strongly relies on the atomic test structure; such a generation process must comply with the constraints imposed by the on-line test environment. Therefore, in this phase the faults not belonging to the easy-to-cover category are considered, only; several test generation strategies can be adopted to identify the

values to complete the atomic blocks and create a suitable program, as described in the next subsections. The goal is to reach the highest possible fault coverage, considering the cumulative effect of the test programs resulting from phases 1 and 2. As shown in the experimental results, the coverage on the complete fault list may reach up to more than 90%.

3. Refinement

At this point of the generation flow, the program obtained during phase 1 is removed. The coverage of the single program developed during phase 2 is evaluated; what is normally observed is a slight decrease in the fault coverage level, but this is usually not substantial, since most of the easy-to-cover faults are detected by both programs.

Starting from this new fault list, the refinement process is another generation step that tackles the few faults not yet detected by the on-line oriented program produced in step 2.

We underline that the major novelty of the proposed approach lies in the introduction of the atomic block and in the adoption of a 3-steps test generation method, while the techniques used in each step are not crucial.

4.2.2 Address Calculation adder SBST experimental results

4.2.2.a Case studies

In order to verify the effectiveness of the proposed approach we performed our experiments on two 32-bit pipelined microprocessor case studies:

1) an automotive microcontroller by STMicroelectronics
2) a demonstrative case based on the miniMIPS core [**93**].

Following the described flow, initial constraints were defined in both the presented cases. Firstly, it was assumed that the size of the memory devoted to allocate the on-line testing programs for the Address Calculation unit is 4KB (2KB for code and 2KB for data). In addition, realistic spaces for code and data memories were also defined for the specific test purpose of the considered unit.

Special considerations have to be done regarding the atomic block selection methodology, since it is possible to adopt different strategies to identify the number of atomic blocks composing the on-line test program, as well as the most suitable values for the operands involved in every atomic block. In the presented cases we adopted two strategies briefly described in the following:

- *Evolutionary optimization tool*: it is possible to use an evolutionary optimization tool such as the one described in [**99**]. The evolutionary tool

can include the atomic blocks as a building element to assemble the test programs. In this case, the evolutionary optimizer can fine-tune the constrained and unconstrained random values in order to maximize every atomic block testing capacities.

- *Random*: a random strategy can utilize the proposed atomic block as an assembling structure where to put the constrained and unconstrained values. A test program can be generated out of a predefined number of atomic blocks.

Special care needs to be taken for fault coverage assessment. In [**50**] the authors describe the method used in this work to evaluate the effectiveness of a SBST test set: this step is very critical, since it has to provide a fair coverage evaluation while taking into account the observability constraints in the on-line test application context, and fault simulation can be a very intensive computational task.

The first case study is a SoC including a 32-bit pipelined microprocessor based on the PowerArchitecture™. It is employed in safety-critical automotive embedded systems, such as airbag, ABS, and EPS controllers. This device contains over 2 million logic gates, including a 576 KB code Flash memory (2 KB of this memory are devoted to store the Address Calculation unit test program) and a 128 KB data RAM (a contiguous 2 KB space is reserved within it for the Test Memory). It also includes other modules, such as an interrupt controller, different buses and I/O interfaces, and a debug controller. Address calculation is performed within the ALU, where two parallel adders are included in the same unit; using different ports, they are in charge of performing both arithmetic operations and address calculations. This module counts 689 gates and 4,188 stuck-at faults; an additional difficulty is that faults in the module affecting arithmetic or address calculations cannot be distinguished.

We applied the generation flow described in the previous section including 1) an easy-to-cover fault removal consisting in a loop-based SBST strategy [**94**] mainly devoted to cover arithmetic adder faults, 2) an evolutionary approach exploiting the μGP3 tool [**99**] that only includes a macro implementing the atomic block described in section 4.2.1.b , and 3) a refinement phase, again resorting to the same tool and considering corner cases and operating on the full fault list. The progression in the fault coverage values for stuck-at faults is shown in TABLE II. Interestingly, the final test program counts only 31 atomic blocks, each composed of 13 instructions.

TABLE II STUCK-AT FAULT COVERAGE [%] OBTAINED ALONG THE FLOW

Case study	**Easy-to-cover fault removal**	**Test program generation**		**Refinement**
		Phase 2 only	*Cumulative (phases 1+2)*	
1	56.67	86.66	91.79	95.11
2	58.02 (59,51)	81.76 (82.85)	88.09 (90.72)	94.27 (96.38)

The second case study is based on miniMIPS [**93**], a processor core based on the MIPS ISA. It features 32-bit buses for data and addresses, and includes a five-stage pipeline. It was synthesized using Synopsys Design Compiler and an in-house developed library, resulting in 33,117 logic gates. In this case, the adder performing the address calculation is a clearly separated unit within the *EXE stage*, counting 342 logic gates and 1,988 stuck-at faults; both address calculation and arithmetic adders count 757 gates and 4,408 stuck-at faults.

The same 3-step strategy was used in this case. However, we employed a random approach for the atomic block selection operation. The results are also shown in TABLE II, where fault coverage values are reported for both adders as well as for the address calculator adder alone (in parenthesis). For the miniMIPS case, the atomic block was reduced to only 6 instructions, and the final program counts 65 atomic blocks.

The proposed methodology is exploited resorting to two different generation strategies, an EA- and a random-based one. Remarkably, in both cases good coverage results were obtained, and the final test programs take about 1.6KB, and require about 800 clock cycles to execute.

In order to corroborate the importance of the selection of the test memory placement, we performed a new experiment using a variable number of atomic blocks, and different code memory allocation constraints: the entire unconstrained processor addressing space, the 2KB space starting at address `0x00000000`, and a configuration especially selected for on-line test, i.e., the address range `0x20007C00-0x200083FF`. TABLE III reports the obtained coverage results for different configurations , before refinement.

It can be observed that a careful selection of the memory space attains better coverage results, and that a small (2 KB) Test Memory allows achieving a fault coverage comparable to that achievable having the whole memory accessible (which is hardly the case for on-line test).

TABLE III STUCK-AT FAULT COVERAGE FOR DIFFERENT CONFIGURATIONS
CASE STUDY 2

Atomic blocks	**Memory allocation constraints**		
[#]	*entire memory*	*2K beginning*	*2K selected for on-line*
8	80.58 %	69.57 %	77.11 %
16	82.24 %	72.03 %	80.68 %
24	82.95 %	72.59 %	81.54 %
32	83.15 %	73.26 %	82.85 %

4.2.3 Conclusions about the SBST of the Address Calculation adder

On-line test application poses a number of constraints to the SBST approach for microprocessor-based systems. The most critical aspects related to the problem were reviewed, and a new methodology for the generation of test programs for Address Calculation circuitry was proposed. The method is mainly based on the adoption of an atomic block, which can be replicated several times in the test programs; the choice of the parameters for each atomic block can be performed using different techniques. Experimental results obtained on both an industrial and a representative case study demonstrates the efficacy of the approach under on-line constraints.

4.3 Register Forwarding and pipeline interlocking unit

When the result of a previous instruction is needed in the pipeline before it is available, a *data hazard* occurs. Register Forwarding and Pipeline Interlock (RF&PI) are mechanisms suitable to avoid data corruption and to limit the performance penalty caused by *data hazards* in pipelined microprocessors. *Data hazards* handling is part of the microprocessor control logic; its test can hardly be achieved with a functional approach, unless a specific test algorithm is adopted. In this section we analyze the causes for the low functional testability of the RF&PI logic and propose some techniques able to effectively perform its test. In particular, we describe a strategy to perform Software-Based Self-Test (SBST) on the RF&PI unit. The general structure of the unit is analyzed, a suitable test algorithm is proposed and the strategy to observe the test responses is explained. The method can be exploited for test both at the end of manufacturing and in the operational phase. Feasibility and effectiveness of the proposed approach are demonstrated on both an academic MIPS-like processor and an industrial System-on-Chip based on the PowerArchitecture™.

A d*ata hazard* is defined as a situation where a processor pipeline produces a wrong output due to data dependency relations between instructions [**100**]. This may happen when the input of one instruction coincides with the output of a

previous instruction, and this output has not yet been written into the proper register when the instruction execution phase takes place (i.e., the two instructions have data dependence). Experiments performed on a MIPS-like pipelined processor with programs from the SPEC92 benchmarks [**101**] show that almost 50% of the executed instructions have some kind of data dependence [**102**]. There are a number of standard techniques developed to work around this situation. It can be handled at compiling time, by suitably inserting no-operation (NOP) instructions between the two instructions with data dependence. However, this leads to larger program binaries and much slower programs. Hence, a common strategy to deal with data hazards reducing executing time penalties is to handle them by hardware, relying on two basic mechanisms: Register Forwarding (or data bypass) and Pipeline Interlock (RF&PI); usually, for optimizing the processor performance, both mechanisms are implemented.

A fault present in the hardware structures implementing the Register Forwarding and Pipeline Interlock mechanisms may result in their malfunctioning and therefore in the processor producing a wrong output; alternatively, the fault may cause an unneeded pipeline stall and consequently a performance penalty. Hence, some of the faults affecting the RF&PI logic fall in the class of performance faults [**103**].

In order to detect these faults (both at the end of the manufacturing process and during the operational phase), a suitable test strategy is needed. The use of approaches based on Design for Testability (DfT), such as scan test or Built-in Self-Test (BIST), is not always possible (e.g., because the processor low-level description is not available or cannot be modified); moreover, some of these approaches are not suitable for at-speed testing, require an expensive external tester, or may not be adopted by designers because they put at risk the IP protection; DfT-based approaches are also not always suitable to support on-line test, which is increasingly often required for safety-critical applications; finally, they may not represent the optimal solution (they have some area overhead, that may lead to extra power consumption and often produce over testing). For these reasons, functional approaches, such as Software-Based Self-Test (SBST), are sometimes preferred [**21**]. SBST consists in uploading into the processor memory a sequence of instructions (the test program), then forcing the processor to execute them, and finally looking at the produced results (e.g., in terms of result values written in memory). The test program should be capable of thoroughly exciting possible device faults and propagating the fault effects to some observable points. This technique does not request any circuit modification; therefore, it is especially suitable for the test of third-parties cores. Moreover, this kind of test is (by definition) performed at-speed, allowing the

capture of frequency-dependent defect mechanisms that arise from process complexities. Finally, test programs generated following the SBST approach may be well-suited to be used for on-line testing.

While SBST presents a considerable number of advantages in certain contexts, developing suitable test programs for some critical blocks (such as the RF&PI one) may be really hard. Both the excitation of the block and the observation of the effect of possible faults are not straightforward.

In this work, we propose a new SBST strategy to test the hardware mechanisms that handle data hazards. In the following, these mechanisms are first analyzed, describing common principles present in different implementations. An algorithm to develop a suitable test program is then provided. Different strategies enabling to observe the faulty behaviour in the forwarding unit and the interlock unit are also discussed, highlighting their main characteristics.

4.3.1 Data hazards and pipeline interlock mechanisms

4.3.1.a Microprocessor behaviour

Register Forwarding (or data bypass) and Pipeline Interlock are functions managed by some combinational logic units included in a pipelined microprocessor to avoid data hazards.

The methods employed to avoid data hazards mainly consist in checking if there is a data dependency between two instructions simultaneously present in different pipeline stages, and take suitable counteractions accordingly. Typically, when an instruction enters the pipeline, the system checks if its input operands correspond to the same register which any other instruction already present in the pipeline is using as output. If this is true, there is a data dependency between the two instructions and some actions have to be taken. For clarity purposes, let us call the first instruction to enter the pipeline *instruction 1* and the instruction arriving later, which has a data dependency with the first one, *instruction 2*. In this case Register Forwarding must be activated: the input data for *instruction 2*, instead of coming from the register file, is directly taken from the stage where *instruction 1* produces it. In case *instruction 1* is not yet in that stage, Pipeline Interlock is used. Pipeline Interlock implies the pipeline is stalled until *instruction 1* reaches the stage in which the data is produced. At that moment, Register Forwarding is used to send the result of *instruction 1* to the stage where *instruction 2* needs its operands.

According to this working mechanism there are different possible forwarding paths (shown in Figure 4.13 for a MIPS-like processor) and their activation depends not only on the existence of some data dependency between two

instructions but also on the stage in which the instructions produce/require the data.

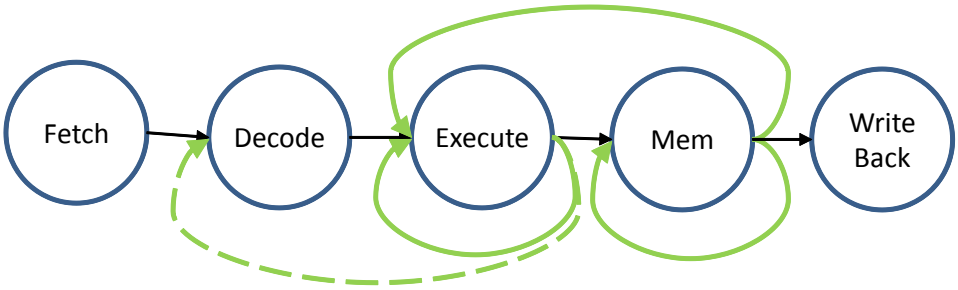

Figure 4.13 Graph of the possible forwarding paths between pipeline stages.

Interestingly, forwarding paths may vary according to the processor architecture. For example, if the branch instructions are completely resolved during the *DECODE stage*, the processor core may include forwarding paths able to feed the *DECODE stage*, as represented by the dashed line in Figure 4.13, and shown as input of de *DECODE stage* in Figure 4.14.

To implement the described working mechanism, the different stages of the pipeline and the Register File module interact with the Register Forwarding and Pipeline Interlock unit within the Data Hazards handling module, as shown in Figure 4.14 using the MIPS architecture as an example.

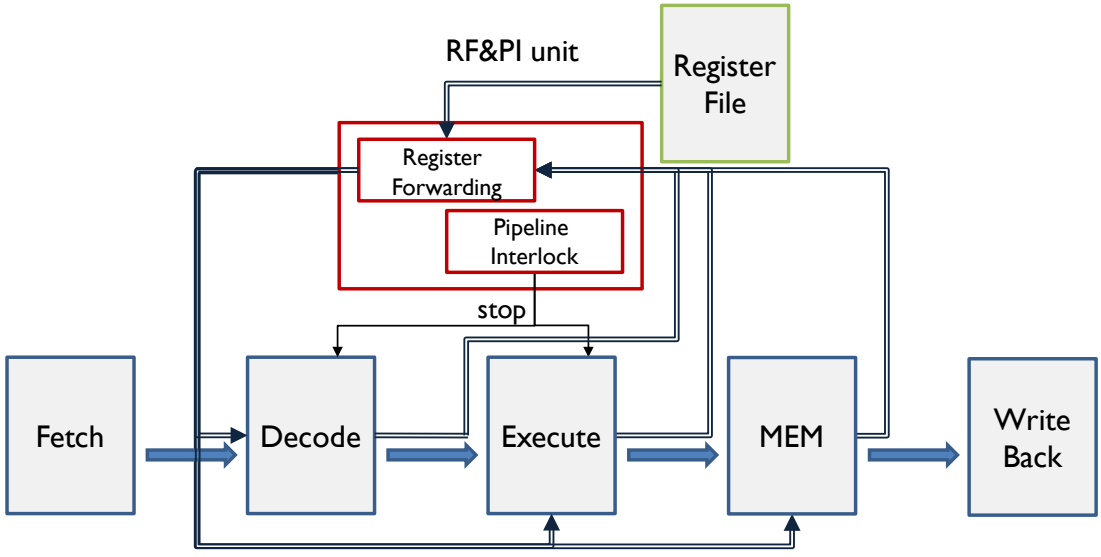

Figure 4.14 The Register RF&PI unit and its interaction with the processor pipeline and Register File module.

Due to the required behaviour of the unit and to its architecture, usually located within the pipeline control logic, specific sequences of instructions only are able

to trigger its action. Obtaining the full activation of the module, required for its test, is a relatively difficult task; also, due to the intensive interaction of the unit with the register file, proper register initialization is needed to build an effective test program.

4.3.1.b Microprocessor architecture

Let us analyze the architecture of the data hazards handling mechanisms, dividing it in the three structural sub-blocks shown in Figure 4.15.

Multiplexers: A multiplexer (or MUX) is a combinational block able to select one input out of multiple ones and connect it to the output signals. Dedicated selection wires indicate which input should be used. The number of selector wires (s) depends on the number of inputs (n) of a MUX; in general (1.1).

$$s = ceil(\log_2 n) \tag{1.1}$$

The *Register Forwarding* (also called *Data Bypass*) logic uses a number of multiplexers to send the appropriate data to each stage. As many MUXs are used as possible destinations exist (e.g., one for operand 1 of the *EXE stage* and a second one for operand 2 of the *EXE stage*, etc.), each one with a number of inputs equal to the number of possible data origins (register file, *MEM stage*, *EXE stage*, etc.).

Comparators: Comparators (CMPs) are mainly used for two purposes: first, to detect if there is a data dependency present in the pipeline, i.e., if two instructions present concurrently in different pipeline stages use the same registers as input and output, respectively; in this case the CMPs compare register identifiers (each register owns a specific identifier which is usually encoded in the instruction operating code). The other use is within the pipeline interlock mechanism to detect potentially unresolved hazards: when an instruction has a data dependency, they check if that dependency is resolved in a later stage than the one when the data is needed and accordingly halt the pipeline when necessary.

Bonding logic: it integrates the logic controlling all the data hazard avoidance mechanisms, determining at any given moment the activation of register forwarding and pipeline interlock based on the processor status.

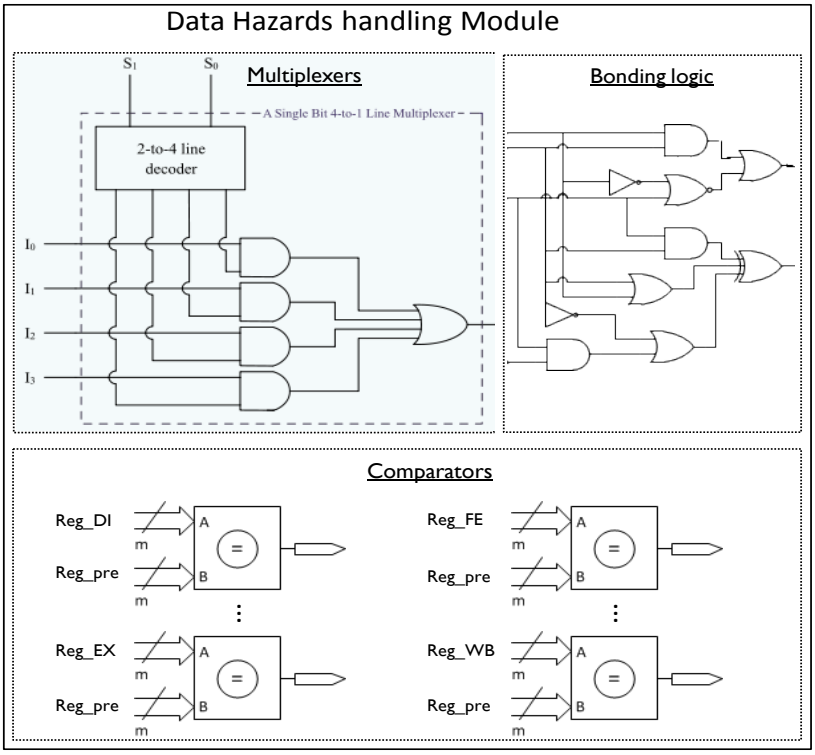

Figure 4.15 Data hazards handling module schema with 3 sub-blocks.

4.3.2 Proposed methodology to test the RF&PI unit

First of all, the reader should note that this unit can hardly be stimulated while testing the other parts of the processor: results reported in the next section experimentally prove that. Intuitively, this is mainly due to the fact that its test requires special sequences of instructions characterized by:

- data dependences between them,
- corresponding register operands,
- suitable observation mechanisms.

The test algorithm routine is developed bearing in mind that the RF&PI unit is mainly composed of MUXs and CMPs. proper test algorithms for these types of modules have been considered in [**104**] and [**90**], respectively.

The main contribution of the proposed approach lies in providing a method for generating test programs which are guaranteed to apply the sequence of values mandated by the above mentioned algorithms to each MUX or CMP embedded within the architecture of the addressed unit, and making the produced results observable in a functional manner. The testing of the bonding logic is done as a consequence of applying the testing strategies of the multiplexers and comparators. In this work only address stuck-at faults were addressed.

4.3.2.a Test of the multiplexers

In [**104**] the authors consider a number of different implementations for a MUX, and prove that they can all be tested by the same set of input vectors. In particular, they prove that when considering a generic n-to-1 multiplexer a set composed of *2n* test patterns can achieve full stuck-at fault coverage.

The set of necessary test vectors for an 8-to-1 MUX with 1 bit parallelism (i.e., 8 inputs and 1 output, each one composed of 1 bit) is reported in Table I S_i denotes a selection signal, D_i an input data signal, while Z is the output.

TABLE IV TEST VECTORS FOR A 8-TO-1 MUX

Vector No.	S_2	S_1	S_0	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	Z
0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	1	0	1	0	0	0	0	0	0	1
2	0	1	1	1	1	1	0	1	1	1	1	0
3	0	1	0	0	0	1	0	0	0	0	0	1
4	1	1	0	1	1	1	1	1	1	0	1	0
5	1	1	1	0	0	0	0	0	0	0	1	1
6	1	0	1	1	1	1	1	1	0	1	1	0
7	1	0	0	0	0	0	0	1	0	0	0	1
8	1	0	0	1	1	1	1	0	1	1	1	0
9	1	0	1	0	0	0	0	0	1	0	0	1
10	1	1	1	1	1	1	1	1	1	1	0	0
11	1	1	0	0	0	0	0	0	0	1	0	1
12	0	1	0	1	1	0	1	1	1	1	1	0
13	0	1	1	0	0	0	1	0	0	0	0	1
14	0	0	1	1	0	1	1	1	1	1	1	0
15	0	0	0	1	0	0	0	0	0	0	0	1

When moving to MUXs having a parallelism greater than 1, the same set of vectors can be used, substituting the 1/0 value with as many 1/0 bits as the parallelism is.

We now want to transform the above set of input vectors into a sequence of instructions to be executed by the processor able to:

- apply the same set of values to the generic MUX within the RF&PI unit,
- make the MUX output observable.

A specific MUX channel is activated when a data dependence between two instructions is detected, and the result produced by the first (i.e., stored in a given pipeline register) must be forwarded to the second (i.e., to the input of a given stage). Hence, the considered MUX inputs are the data registers existing within the pipeline registers. This means that for every MUX in the RF&PI unit (i.e., for each possible data dependence configuration) a program fragment can

be created, including a couple of data dependent instructions that trigger the Register Forwarding. Ideally, the fragment should be replicated as many times as the number of test vectors required to test the MUX. For each replica the values of the pipeline registers corresponding to the inputs of the MUX should hold the all 0s or all 1s value, according to what mandated by TABLE IV. Finally, the fragment should include an instruction to make the output of the MUX observable.

Let us explain this technique with a simple example oriented to apply the proposed approach to an eight-bit wide 3-to-1 MUX for feeding the first operand of the *EXE stage* in a pipelined processor. Assuming that the processor stages are fetch (FE), decode (DE), execute (EXE), memory (MEM), and write back (WB), as described in [**100**], and the considered operands are provided to the MUX inputs from DE, when no forwarding is required, and from EXE, and MEM stages, when forwarding is needed.

Let us suppose that the first MUX input comes from the *DECODE stage*, the second one from the *EXE stage*, and the last one from the *MEM stage*. Then, as an example, we report an assembly-like sequence that could be used for applying the first two vectors proposed in [**104**], and reported in TABLE V.

TABLE V INPUT VALUES FOR THE 4-TO-1 MUX FEEDING THE FIRST OPERAND INPUT OF THE *EXE STAGE* IN A PIPELINED PROCESSOR

Vector No.	S1	S0	D0(DE)	D1(EXE)	D2(MEM)	Z
0	0	0	00	FF	FF	00
1	0	1	00	FF	00	FF
			…	…	…	
⋮	⋮	⋮	…	…	…	⋮
			…	…	…	

TABLE V reports the required input values in the MUX under test. As the reader can notice, the different values in the MUX inputs at every clock cycle depend on the instructions that traverse the processor pipeline during the considered instance of time. The program in Figure 4.16 reports a sequence of instructions able to apply the mentioned vectors in the MUX under evaluation.

Assuming that there are no cache misses and that the LD/SD instructions require only one clock cycle during the *MEM stage*, the first vector in TABLE V is applied during the *EXE stage* of instruction 5. In the considered clock cycle, the input value of the MUX under test is actually provided by the *DECODE stage* that reads from the register file the value of `r2`, already set by instruction 2. Instructions 3 and 4 propagate through the pipeline the rest of the values (`0ffh` for both cases)

required by the test vector. Additionally, instructions 1 and 9 are devoted to add observability capacities to the considered code fragment.

```
1.    add   r1, r0, 0ffh      ;prepare the register for observability
2.    add   r2, r0, 00h
3.    lw    r3, FF_val(R0)
4.    add   r4, r0, 0ffh
5.    add   r1, r2, r0        ; vector 0
6.    nop
7.    nop
8.    nop
9.    sw    r1, data1(R0)     ;observability instruction
                              ;r1 value is ready for observability
10.   lw    r3, 00_val(R0)
11.   add   r2, r0, 0ffh
12.   add   r1, r2, r0        ;vector 1
13.   nop
14.   nop
15.   nop
16.   sw    r1, data2(R0)     ;observability instruction
```

Figure 4.16 Test program fragment for testing the MUX for the *EXE stage*.

The second vector is applied by the block of instructions 10-16. In details, instruction 12 depends on instruction 11 that forwards its output value (the one for r2) from the output of the *EXE stage* to the input of the MUX under evaluation. Once again, the rest of the instructions set appropriate values in the pipeline and support observability.

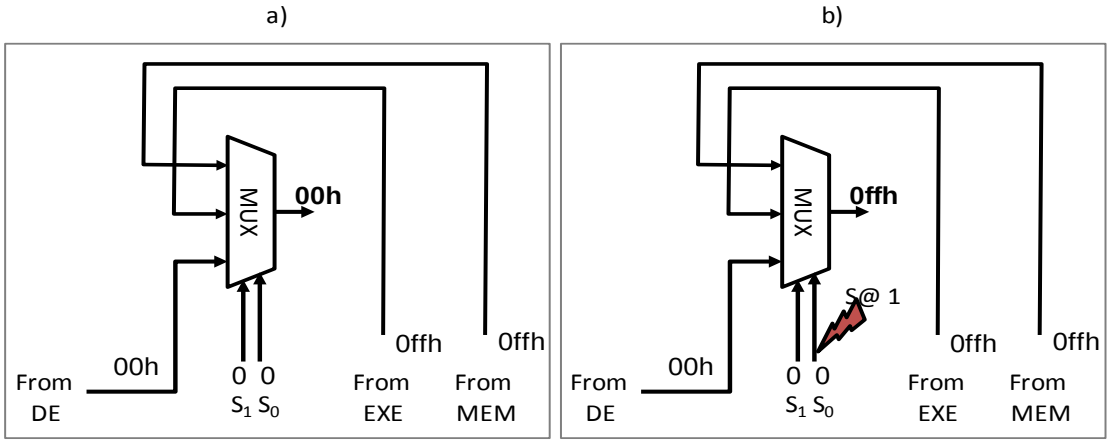

Figure 4.17 a) Normal and b) Faulty behaviour in one selector.

Figure 4.17 a) describes the normal behaviour for the sample program at the time in which vector 0 is applied. The different values are propagated exploiting the block of instructions 3-5 that guarantees the values 00h, 0ffh, and 0ffh in the MUX inputs.

If a structural fault occurs in one of the selection wires, as in Figure 4.17 b), the MUX outputs assume a different value (in this case `0ff`h) easily detected thanks to the observability instructions. Similarly, if a stuck-at fault is located in one of the forwarding paths, the error effect will result in a MUX output different from the expected `00h` (or `0ffh`).

The reader can notice that the rest of the patterns of TABLE IV (vectors 2-15, in the case of an 8-to-1 MUX) can be easily translated to assembly instructions following the same scheme provided for vectors 0 and 1. With this algorithm we assure a total coverage of faults in the decode logic of multiplexer, with an optimum scalability in terms of bit-wise parallelism and a short SBST code footprint.

4.3.2.b Test of comparators

To thoroughly test an m-bit wide comparator (i.e., 2 inputs, with m lines each), independently on its low-level implementation, it was stated in [**90**] that one can use a set of 2m+2 patterns. In the same article it is shown that the following patterns allow achieving complete fault coverage.

Out of the 2m+2 vectors, two correspond to the situation in which the two CMP inputs match: this means that all the corresponding bits in the two input operands are equal. Each bit holds an opposite value in the two vectors.

Each of the other 2m patterns generates a mismatch in only one bit of the pair of words fed in parallel into the comparator. Hence, this set of patterns corresponds to a walking 1, starting from the MSB of one of its inputs going all the way to the LSB of the other input. In Figure 4.4 we already showed the schema of a comparator, and the patterns to be applied to test it, and it was highlighted that, being combinational logic, the order in which these patterns are applied is irrelevant.

Once more, the proposed approach aims at developing a program fragment able to apply the proper set of test vectors to each CMP in the RF&IP unit. As already mentioned, the detection of data dependencies in the pipeline is done with comparators. For this use there is one comparator per input operand, per each stage involved in the RF&PI mechanism. In common processor architectures having two operands per instruction this means a series of comparators that check the two operands against the possible sources in every one of the pipeline stages where these values may be produced (see Figure 4.13). Accordingly, one of the inputs of these comparators is connected to the operand identifier (i.e., a register) encoded in the instruction and used by the instruction in one stage, while the other input is the identifier of the output register on one of the possible source stages where the other instruction in the pair assumed to have a data

dependence is placed. The same applies for all relevant pipeline stages. In this way any data dependency can be detected. In order to excite these comparators with the required patterns, registers with identifiers following the patterns previously described should be addressed by consecutive instructions.

The example in Figure 4.18 shows a piece of code intended to excite the comparator for operand 1 in the *EXE stage*, considering the processor architecture described in section 4.3.1.b . One input of the comparator is connected to the pipeline register storing the identifier of the output register of the instruction in the *EXE stage*; the other input is connected to the pipeline register storing the identifier of the operand of the following instruction. The red arrows show the performed comparisons. For this example let us use a register file composed of 8 registers, so that 3 bits are used to represent the register identifier; consequently, we need to perform 8 comparisons to apply the sequence of test vectors proposed above.

The used registers must be properly initialized, in order to make possible faults observable. In particular, please note that r0 and r7 are initialized with values different from all the other registers. Similarly, the value for k should be carefully chosen to avoid any overflow (e.g., k=0).

The proposed sequence of 17 instructions thoroughly excites the comparator of the example. The only purpose of the nop instructions is to create the necessary distance between the relevant instructions.

The same sequence of instructions can be easily tailored to test the other comparators. For example, it can be modified to target both comparators related to the *EXE stage*, just using an instruction with two input operands (e.g., add `rA`, `rB`, `rC`; `rA` ← `rB` + `rC`) instead of an immediate operand. Additionally, placing a useful instruction (for testing purposes) instead of the nop instruction can allow us to also test the comparators of other stages at the same time. By generalizing this solution, the addition of one instruction per involved stage allows testing all comparators.

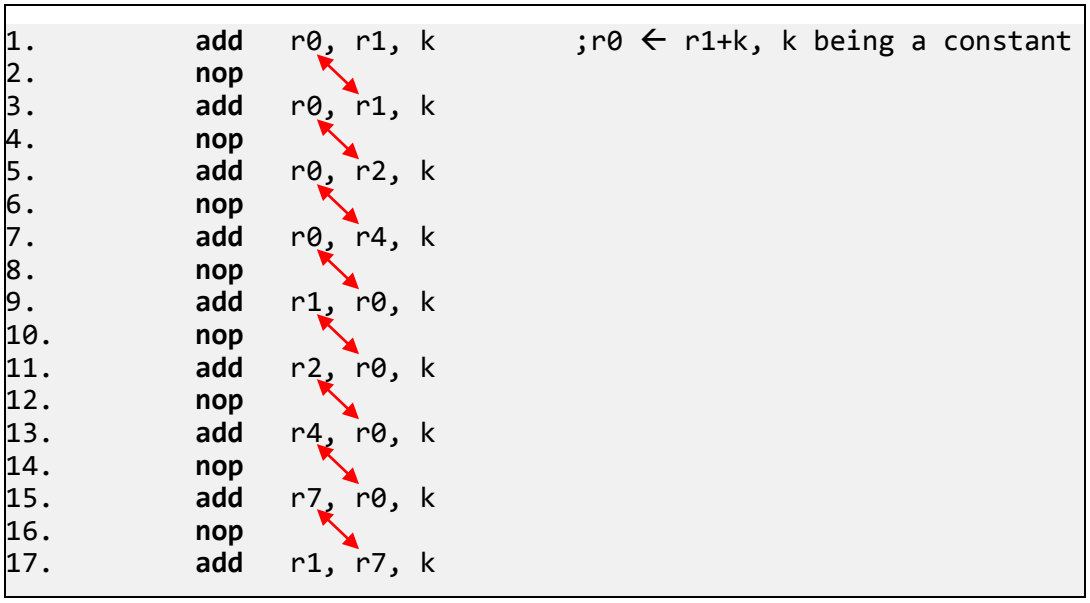

Figure 4.18 Test program fragment for testing the *CMP* in the *EXE stage*.

Another use of comparators within the data hazard mechanism is in the pipeline interlock activation. Let us define the level of an instruction as the stage in which the data produced by that instruction is ready, or when the effects of the instruction are seen; for example, in the miniMIPS architecture the level of an ADD is the *EXE stage*, while the level of a LOAD from memory instruction is the *MEM stage*. This information is hardcoded within the microprocessor and gathered when each instruction is decoded (i.e., at the *DECODE stage*). When a data dependency is identified, the level of the instruction producing the result to be forwarded is compared with the stage in which the instruction is, in order to identify potentially unresolved hazards. In the case of the data hazard handling module, there is one comparator per stage devoted to this task, and its inputs are connected to a constant indicating the stage itself and to a signal indicating the level of the instruction present in that stage.

4.3.2.c Observation mechanism

In the previous subsections we described how to write test programs able to test the RF&PI unit. This test can be performed by suitably activating the different components of this unit, and then making the produced results observable.

However, some of the faults affecting the RF&PI unit may not produce any wrong results, but rather a change in the performance of the processor, e.g., by introducing unnecessary stalls. For the purpose of detecting these faults, we thus need to exploit some mechanism to measure the time required to execute the test program (or some of its parts).

This can be accomplished resorting to the same strategies proposed to observe the results when testing the BTB in section 4.1.4.b :

- using the performance counters [**91**] existing in many processors and able to count the number of stalls,
- resorting to some timer able to measure the performance of the processor when executing a given piece of code,
- adding some ad hoc module to the system able to monitor the bus activity [**92**].

4.3.3 RF&PI unit SBST experimental results

4.3.3.a Case studies

The feasibility and effectiveness of the proposed approach have been evaluated on an academic processor module and on a commercial microprocessor.

The selected academic case study is the miniMIPS processor [**93**] synthesized using an in-house developed library and resulting in a 16,236-gate circuit (without multipliers). The miniMIPS architecture is based on 32-bit buses and includes a five-stage pipeline. The RF&PI unit occupies around 3.4% of the total number of gates of the processor, accounting for a total of 3,738 stuck-at faults.

In order to experimentally validate our approach, we considered first a test set of programs tackling the whole miniMIPS processor core that achieves about 91% fault coverage with respect to stuck-at faults. The test programs contained into the test set were developed following state of the art strategies such as [**36**]. However, the stuck-at fault coverage achieved on the RF&PI unit reached only about 66%, thus proving that specific test algorithms are required for it.

Secondly, we developed a test program (RF&PI TP) following the algorithm described in the previous section and specifically targeted to the RF&PI unit. Its main characteristics are summarized in TABLE VI in terms of size, duration, and stuck-at Fault Coverage on the RF&PI unit. The few untested faults (39 out of 3,520) are mainly related to signals and features that cannot be activated using a functional approach, such as the interrupt signal; other faults remain untested because the module can support a coprocessor, which was not used in our experiments.

TABLE VI CHARACTERISTICS OF THE TEST PROGRAM FOR THE RF&PI UNIT

	Size [bytes]	**Duration [clock cycles]**	**%FC (RF&PI)**
RF&PI TP	4.236	2.084	98.89

We also applied the proposed technique to a commercial System-on-Chip including a 32-bit pipelined microprocessor based on the Power Architecture™ and manufactured by STMicroelectronics. The device contains over 2 million logic gates and is employed in safety-critical automotive embedded systems, such as airbag, ABS, and EPS controllers. For this reason, a suitable test which can be applied even in the operational phase is mandated by the ISO 26262 standard [**7**].

In this case, the module playing the role of the RF&PI unit accounts for about 14K faults. The functional test program developed for the whole circuit reached only about 62% stuck-at fault coverage on the RF&PI unit. After adding to it some further test fragments developed according to the proposed test algorithm (and accounting for about 2,000 instructions and less than 5,000 clock cycles), the fault coverage on the same unit raised to about 92%. Obviously, implementation of the proposed algorithm on this architecture is harder than for the miniMIPS, due to the higher complexity of the pipeline and the higher number of functional units.

For both considered circuits the logic simulation was performed using ModelSim SE 6.4b by Mentor Graphics, while the Fault Coverage was evaluated using TetraMax v. B-2008.09-SP3 by Synopsys.

4.3.4 Conclusions about the RF&PI unit SBST

The chapter proposes a strategy for the functional test of the mechanisms for Data Hazards handling and Register Forwarding in pipelined microprocessors. We first showed that the module implementing these mechanisms could hardly be tested in a functional manner without explicitly targeting it. The typical architecture and behaviour of this module were detailed, and a test algorithm was proposed, able to fully exercise and observe the target unit. The proposed approach is fully functional (i.e., it does not rely on any DfT infrastructure) and does not rely on detailed information about the implementation of the addressed unit.

Preliminary results were obtained considering the stuck-at fault model, both for an academic case study and an industrial one. They experimentally prove that the proposed technique is effective in achieving high fault coverage on the target module.

Chapter 5
Proposed Infrastructure-IP to augment self-testing capabilities

This chapter proposes an alternative solution that still falls in the SBST domain, since it is based on instruction execution, but also exploits some concepts borrowed from the BIST field; in particular, some system reorganization is adopted during test, and the instructions executed by the processor are generated on-chip when in test mode.

The technique we propose forces the processor to execute a compact SBST-like test sequence by using a hardware module called MIcroprocessor Hardware Self-Test (MIHST) unit, which is intended to be connected to the system bus like a normal memory core, requesting no modification of the processor core internal structure. The benefit of using the MIHST approach is manifold: while guaranteeing the same or higher defect coverage of the traditional SBST approach, it reduces the time for test execution, better preserves the DUT Intellectual Property (IP), does not require any memory to store the test program, and can be easily adopted for on-line testing, since it minimizes the required system resources. The feasibility and effectiveness of the approach was evaluated on the miniMIPS processor, used to test both the processor itself and embedded memory cores.

5.1 MIHST – A new Hardware-Based Self-Test concept

SBST is an appealing alternative to perform self-test of both microprocessors and embedded memories [**57**][**58**]. Unfortunately, the SBST approach shows some limitations, in particular in the on-line scenario

- Some faults modify the test program flow and potentially lead to an endless execution (i.e., caused by a wrong jump outside the test code portion), making difficult to take back the control of the system at the end of the test
- Some memory addresses are never accessible because not reserved to the test procedure, therefore resulting in a coverage loss

- Size and execution time of the test could be prohibitive in case of stringent real-time application requirement
- IP protection is not guaranteed, since the test program may reveal details about the processor core implementation.

To address the above issues a novel methodology is proposed in this chapter, named MIcroprocessor Hardware Self-Test (MIHST), which is particularly suitable for on-line testing of embedded processors in Systems-on-Chip (SoCs), and overcomes most of the current SBST limitations. The proposed method combines features of hardware-based and functional-based techniques.

The approach basically consists in the insertion of a programmable Infrastructure-IP (I-IP) named MIHST unit, that is connected to the system bus (Figure 5.1).

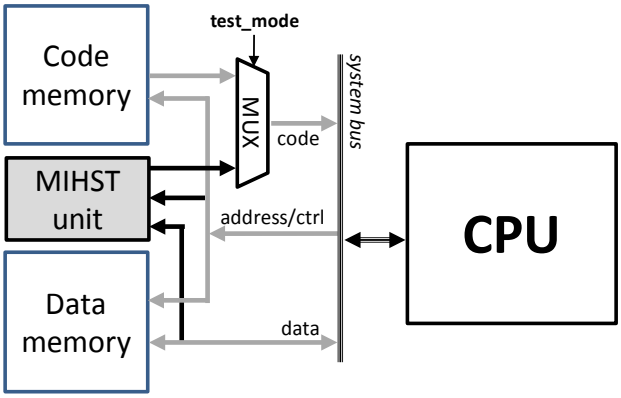

Figure 5.1 Architecture of a system including the MIHST unit

The MIHST architecture is founded on two ideas

1. The system may be either in the normal or in the test state; in the former, the processor executes the instructions read from the code memory; in the latter, the MIHST unit generates an instruction stream for the processor core, substituting the code memory, while also observing the processor behavior. When generating the instruction stream, the MIHST unit does not care about the sequence of instruction addresses generated by the processor for instruction fetch purposes
2. The MIHST unit internally encodes the test program in a custom manner to exploit the test program regularity and to minimize the hardware required to store it.

As a consequence of the above ideas, the processor executes the instructions provided by the MIHST unit, but does not control any more the execution flow: as an example, when the MIHST unit provides the processor with the code of a conditional jump instruction, the processor executes the instruction (i.e., it

evaluates the condition and depending on the result it generates a different address during the following fetch cycle), but the following instruction is decided by the MIHST unit independently of the address generated by the processor.

This working principle is crucial because

a) the test engineer can manipulate the address range at his/her will when designing the test program, thus contributing to a better coverage.
b) it permits to observe the result of an instruction execution on the bus without having the rest of the test procedure compromised by any possible fault.

Roughly speaking, the latter point means that the execution flow of the MIHST driven program is never altered by a fault occurrence. For example, in a pure SBST scenario some faults may lead to an exception raising (e.g., erroneous access to memory). However, this is not the case in a MIHST supported test flow, since the MIHST unit is forcing a predefined instruction stream that does not depend on the processor requests.

The most evident advantage introduced is the reduction in terms of required resources for test (in particular in terms of accessible memory area) with respect to the pure SBST solution.

Furthermore, the MIHST approach allows to significantly shorten the execution time; for instance, instructions devoted to manage the program control flow (i.e., loop constructs) are no longer required, unless they are strictly required by the test procedure.

The MIHST unit can be programmed with a few highly encoded data that can be either uploaded from the outside, or hardwired in it; in both cases, a significant benefit exists with respect to the traditional SBST approach in terms of IP protection (no explicit test program is given to users). Ideally, an IP provider could simply release to its customers the MIHST unit for the processor core (with the encoded test program hardwired in it), and the customers could use it for testing the core.

To demonstrate the feasibility and effectiveness and to estimate costs and benefits of the MIHST approach, the method has been applied to a miniMIPS [**93**] processor-based system and results have been gathered both for the testing of the processor itself as of an example embedded memory.

5.2 MIHST – An embedded microprocessor testing strategy

The proposed methodology for microprocessor testing merges some of the principles of Software-Based Self-Test (SBST) and Built-In Self-Test (BIST). The idea is to adopt an Infrastructure IP (I-IP), called Microprocessor Hardware Self-Test (MIHST) unit, which is connected to the system bus. The interconnection of the I-IP within a processor-based SoC is supported by a multiplexer, as shown in Figure 5.1. The MIHST unit insertion does not require any modification in the processor's core, similarly to BIST approaches [**105**][**106**]. The I-IP only intervenes during the test phase, while being transparent during mission mode.

The MIHST behaviour is mainly based on two principles, as briefly described in the introduction:

- *Forced instruction sequence*: the MIHST unit forces on the bus a sequence of instructions which is not dependent on the addresses generated by the processor
- *Encoded test program storage*: the usual regularity of functional test programs is exploited to store the instruction sequences in a strongly encoded manner which permits saving time and space.

5.2.1 Forced instruction sequence

The MIHST unit is designed to respond to the fetch cycles of the processor by providing instructions through the bus, mimicking a code memory. However, this instruction sequence is stiffly generated by the MIHST, without taking into account the specific addresses produced by the processor as a response to the execution of the previously provided instructions.

This capability is crucial for overcoming some limitations that pure SBST procedures show when executed:

- A MIHST test always finishes
- Address generation is no longer a problem because the processor receives instructions autonomously provided by the MIHST.

Figure 5.2 exemplifies the introduced concept on a sample piece of code (a) and shows the resulting execution flow both in the normal (b) and in MIHST (c) execution mode.

The sample code includes a jump instruction producing K iterations of instructions 3 to 6. In normal mode, the code execution is driven by the addresses emitted by the processor and thus the execution flow jumps back or

forth according to the presence of jump/branch instructions. When the same piece of code is executed by the processor in MIHST mode the sequence of instructions to be executed is autonomously generated by the MIHST unit; assuming that for test purposes the execution of the instructions composing the cycle body is only required once, the execution flow may not include any iteration.

As a further explanation, Figure 5.3 reports the sequence of values hold by the Program Counter (PC) and the Instruction Register (IR) during the mission mode execution of the code fragment in Figure 5.2; by including a value in brackets, we denote the content of the memory cell corresponding to a given address. It can be seen that the PC and IR values are strictly correlated. On the contrary, in the proposed MIHST mode (bottom part of Figure 5.3) this correlation is intentionally lost. The PC value is no more controlling the code flow which is forced by the MIHST unit no matter the address written on the bus during the fetch cycle.

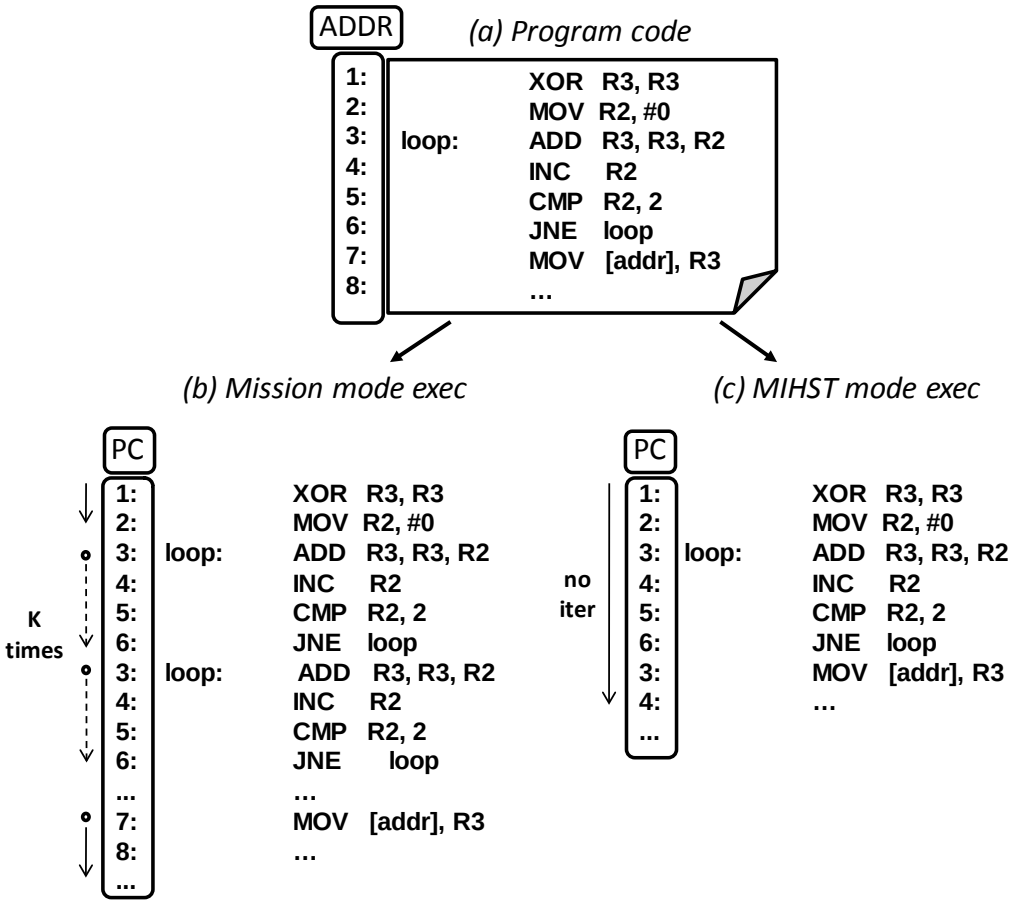

Figure 5.2 Program execution flow in normal and test mode.

MISSION MODE

PC	1	2	3	4	5	6	3	4	5	6	7	8	…	
IR	-	[1]	[2]	[3]	[4]	[5]	[6]	[3]	[4]	[5]	[6]	[7]	[8]	…

MIHST MODE

PC	1	2	3	4	5	6	3	4	5	6	7	8	…	
IR	-	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	…

Figure 5.3 PC and IR evolution in mission and MIHST mode.

During test mode, the PC values are monitored by the MIHST unit by reading them from the address bus together with the other control signals. In case a branch instruction is executed, even if the next code word put on the bus is not determined by the PC, the value written on the address bus by the processor is related to the execution of such branch instruction, and thus observed by the MIHST unit, for example to support the test of possible branch unit faults.

The instruction flow imposed by the MIHST unit can encompass more complex schemas than the explanatory "forced sequential" execution illustrated in Figure 5.2 and Figure 5.3. In our design case, such capabilities are supported by the MIHST unit through an ad-hoc instruction set architecture enabling to control up to 2 levels of nested loops. From the processor functionalities point of view, any forced sequence is valid because of the introduced MIHST working concepts.

Instructions are pumped in the bus according to a flow which is decided by the MIHST unit and not related to the processor internal state. This is critical to improve actual SBST methods, because:

1. It makes the program code structure simpler, since there is no need to respect semantic constraints.
2. Branches, as well as exception handling functionalities are tested by simply spying the bus, without the need for storing a test program distributed over the full memory space.
3. Since the execution flow is not controlled by the processor, instructions such as NOP may be inserted within the pumped instructions to obtain:
    - Correct data memory accesses;
    - Effective methods to test complex features such as forwarding paths and stalls of the pipeline.

5.2.2 Encoded procedure description

In the proposed scenario, the instruction stream produced by the MIHST unit on the bus needs to be generated on the fly; therefore, the MIHST has to produce instructions quite quickly by reading information from some internal memory.

To avoid the insertion of a large memory core within the MIHST unit, an encoding schema has been devised, which permits saving test execution time, too.

The encoding method exploits the usual regularity of test programs written according to the SBST strategy. The key idea is to possibly encode several instructions into a single line, asking the MIHST unit to reproduce the original sequence.

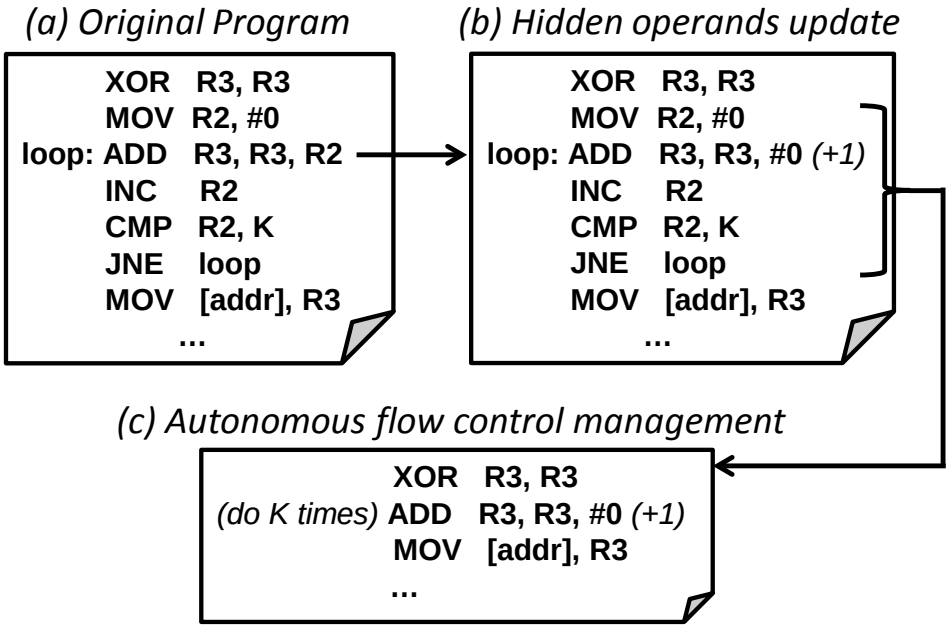

Figure 5.4 Program execution flow in normal and test mode.

In any test program, a target instruction can often be identified which is the core for testing a considered module; this instruction is repeatedly executed with different operands for a determined number of times: as an example, testing the adder resorts to the ADD instruction, while testing a branch prediction unit is typically based on branch instructions. In both cases the ADD or branch instruction is repeatedly executed to thoroughly test the module under test. The encoding method stems from this observation and is practically built on the following actions managed by the MIHST unit:

a) *Hidden operands update*: parametric operands are replaced by immediate values whose evolution is controlled by the MIHST unit itself under the indications provided in the encoded program
b) *Autonomous flow control management*: information about the test program flow are provided to the MIHST unit that takes care of managing the target instruction repetition with different operands with no need of additional instructions.

As an example, let us consider the example provided in Figure 5.2, which can be manipulated as in Figure 5.4; in this case, the loop-based construction of the functional program makes the ADD (target) instruction to be repeated K times with different operands produced by the instruction INC included in the loop. This code can be manipulated according to the hidden operand update and autonomous flow control management principles. First, the ADD instruction is modified in such a way that for every iteration the variable operand is updated *(+1)* directly by the MIHST unit; then, control flow specific instructions are eliminated, since the operand update is directly managed by the MIHST unit (*do K times*).

5.2.3 MIHST unit architecture and behaviour

The MIHST unit operates like a memory core, feeding the processor with values corresponding to assembly instructions, thus enabling the execution of sequences of instructions. Similarly to a usual memory BIST scenario, a multiplexer permits to drive the data bus with instructions coming either from the code memory or from the MIHST unit.

When in test mode, the MIHST unit replaces the code memory, and interacts with the processor repeatedly performing the following operations :

- Detects an instruction fetch cycle
- Reads and decodes instructions from the internal memory
- Generates an instruction code for the processor
- Observe the processor behaviour, compacting the values it produces on the address, data and control signals.

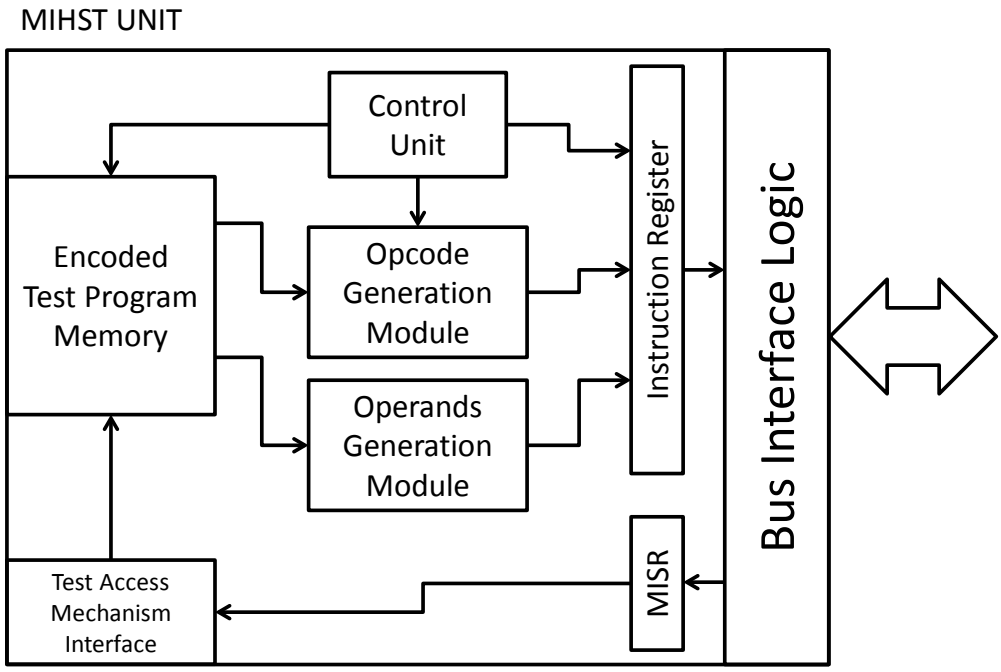

Figure 5.5 Schematic view of the MIHST unit architecture.

Figure 5.5 shows the MIHST unit architecture. Address, Control and Data buses are read by the MIHST unit both to monitor them (in order to detect possible faults) and to comply with the bus protocol when it must write on them.

The architecture of the MIHST unit includes:

- An instruction register, holding the value to be put on the bus when the CPU performs a read cycle.
- A set of internal memories for storing the encoded information about the instructions to be generated, resorting to an ad hoc instruction set including:
  - OPCode words, identifying instructions and including information about their execution under the MIHST unit control;
  - OPErand words, describing operands to be applied and their evolution along the program.
- Two instruction generation modules, namely:
  - the OPCode Generation (OPCG) module, in charge of generating microprocessor instruction operational codes;
  - the OPErand Generation (OPEG) module, in charge of generating instruction operands. It may be replicated more than once, depending on the ISA of the processor under test (i.e., on the maximum number of operands of an instruction); this module

implements the simple manipulations that may be applied to an operand within a loop, such as shift, increment, etc.

- A Control Unit, managing the overall application flow in collaboration with the Bus Interface Unit.
- A Bus Interface Unit, reading and writing the system bus.
- A Results Collection module, compressing monitored address, data and control bus signals; this function is suitable to be implemented by a MISR module.
- An optional Test Access Mechanism module in charge of interfacing the MIHST unit with the Automatic Test Equipment in case the encoded instruction sequence has to be uploaded from the outside. This module does not exist if the test program is hardwired in the MIHST unit, as it is likely when dealing with on-line test.

The format used to store each MIHST encoded instruction is shown in Figure 5.6, where each OPCode field includes the following sub-fields:

- *OPCode*: contains a bit string identifying the processor instruction to be written on the system bus
- *iloop*: this sub-field indicates whether the instruction is the start of an inner loop, and its length
- *oloop*: this sub-field indicates whether the instruction is the start of an outer loop, and it length.

In each OPErand field the following sub-fields exist:

- *operand seed*: contains a seed to generate the operand sequence
- *manip*: each bit in this sub-field corresponds to a logical or arithmetic operator to be applied to the operand seed, e.g., left-shift (<<) and increment (+1). A bit set to 1 means that the corresponding operator is applied; operators are applied in order, starting from the most significant bit in the field. If every bit is set to 0 the operand remains fix.
- *rst*: indicates whether the original seed has to be restored on the first iteration of the inner loop (if there is one), within the outer loop.

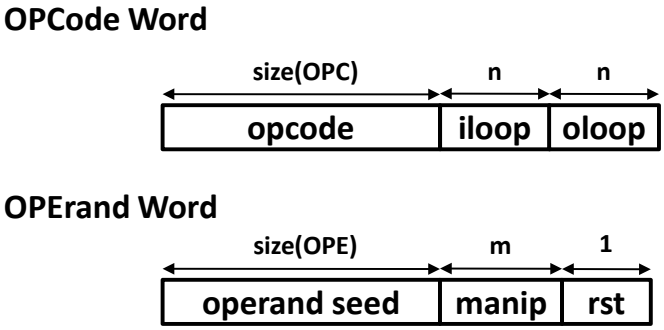

Figure 5.6 MIHST unit instruction encoding.

5.2.4 Encoded instruction generation

The MIHST-encoded instruction sequence can be produced in two different ways:

1. starting from an available test program, intended for SBST for example, and systematically manipulating it according to the concepts described in section 5.2.2 to obtain the compressed MIHST-ready version,
2. directly generating it, profiting from the particular characteristics of the MIHST approach, which allow memory manipulations that would otherwise be impossible.

5.2.4.a Generating the program starting from an SBST program

The compressed code generated for the MIHST unit can be derived from an already existent SBST program through three code manipulation steps which are performed in a preliminary off-line phase. The manipulation steps, schematized in Figure 5.7, are:

1. *Unrolling*: any loop in the original code is unrolled, producing a code with many replicated instructions
2. *Sifting*: besides branch instructions, that are removed during unrolling, other instructions that do not significantly contribute to the fault coverage are eliminated, such as those controlling iteration indexes
3. *Encoding*: by exploiting the regularities of the sifted code, the new code is compressed performing an operation that can be seen as an inverse unrolling transformation.

Steps 1 and 2 shorten the test execution time while maintaining the same coverage than the original code; however, the code footprint becomes too large. Therefore, an encoding phase is performed in step 3 to compress as much as possible the sifted code. The final program is ready to be uploaded in the small memory within the MIHST unit.

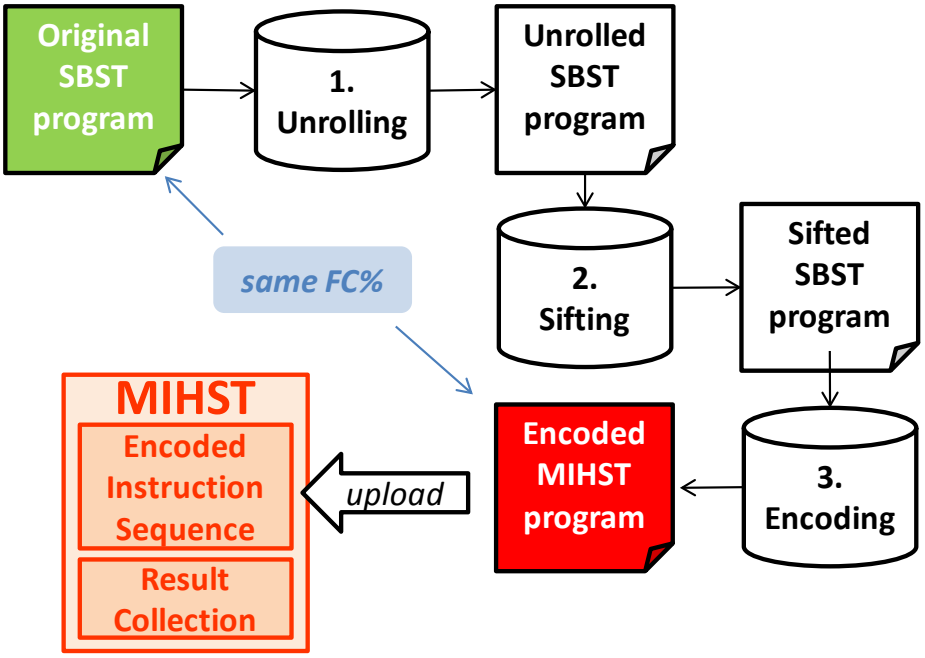

Figure 5.7 Program manipulation flow.

The method allows attaining several advantages when adopted on state-of-the-art test programs. Main benefits are test time and code size reduction, while fault coverage is maintained unchanged.

The example that follows is related to a generic pipelined processor, and code chunks are written in a generic pseudo assembly language.

Example: Register file test module

Let us consider a loop-based approach for testing a Register File similar to the one proposed in [**47**]. The SBST test program that follows is written considering a processor implementing a 4 bits data path, and a 8-entries Register File. The program consists in a sequence of set/reset operations on each register, as shown in Figure 5.8.

The execution of the original SBST procedure requires 78 clock cycles (assuming no stalls), which can be reduced by exploiting loop unrolling. The unrolling operation also allows to remove some instructions (such as those for initializing and decreasing r7 for managing the loop).

```
1.          set   r0, 0000b
2.          set   r1, 1111b
3.          set   r2, 1111b
4.          set   r3, 0000b
5.          set   r7, 2
6.    loop: mov   r4, r1
7.          mov   r5, r0
8.          mov   r6, r3
9.          sw    [RAM], r0
10.         sw    [RAM], r1
11.         sw    [RAM], r2
12.         sw    [RAM], r3
13.         sw    [RAM], r4
14.         sw    [RAM], r5
15.         sw    [RAM], r6
16.         set   r0, 1111b
17.         set   r1, 0000b
18.         set   r2, 0000b
19.         set   r3, 1111b
20.         dec   r7
21.         beq   r7, 0, loop
22.         mov   r7, r0
23.         sw    [RAM], r7
24.         mov   r7, r2
25.         sw    [RAM], r7
```

Figure 5.8 Original Register File module test program.

The execution of the unrolled and sifted version of the SBST program (Figure 5.9) requires 72 clock cycles.

```
1.          set   r0, 0000b
2.          set   r1, 1111b
3.          set   r2, 1111b
4.          set   r3, 0000b
5.          mov   r4, r1
6.          mov   r5, r0
7.          mov   r6, r3
8.          mov   r7, r2
9.          lw    [RAM], r0
10.         lw    [RAM], r1
11.         lw    [RAM], r2
12.         lw    [RAM], r3
13.         lw    [RAM], r4
14.         lw    [RAM], r5
15.         lw    [RAM], r6
16.         lw    [RAM], r7
17.         set   r0, 1111b
18.         set   r1, 0000b
19.         set   r2, 0000b
20.         set   r3, 1111b
21.         --repeat code as in lines 5-16.
```

Figure 5.9 Unrolled and sifted Register File module test program.

Taking advantage of its regular structure, this test fragment can be described in the parametric way reported in Figure 5.10. The operational codes of the selected instructions are stored unchanged, while operands can be constant or parametric values. For example, in the last instruction the operand2 is generated by starting from a seed equal to r0 (register 0) and being manipulated (i.e., incremented r1, r2, r3, etc.) each time the instruction is forced on the bus.

Outer loop	Inner loops	Opcode	Operand 1	Operand 2	Operand 3
2	1	set	r0 (fix)	0000b (+0Fh)	-
	1	set	r1 (fix)	1111b (+0Fh)	-
	1	set	r2 (fix)	1111b (+0Fh)	-
	1	set	r3 (fix)	0000b (+0Fh)	-
	1	mov	r4 (fix)	r1 (fix)	-
	1	mov	r5 (fix)	r0 (fix)	-
	1	mov	r6 (fix)	r3 (fix)	-
	1	mov	r7 (fix)	r2 (fix)	-
	8	sw	[RAM] (fix)	r0 (r:+1)	-

Figure 5.10 Encoded MIHST-ready Register File module test program.

The execution of the sifted SBST procedure and the encoded version account in this case for the same execution time, whereas the code size is strongly reduced.

5.2.4.b MIHST–ready program generation

The code to be executed by the processor following the MIHST approach can also be generated basing on a SBST approach but taking advantage of the flexibility in terms of execution flow that the MIHST approach provides, in order to simplify it. With the introduction of the MIHST unit it is possible to force the processor to execute programs that would be unfeasible to run if the code was stored in the system's code memory.

This flexible behaviour allows to trigger special situations during test mode. As an example, two different instruction codes can be provided by the MIHST unit to the processor when the same value exists in the processor's Program Counter (PC). For example, a branch instruction to the same target address repeated two times may lead to the execution of two different target instructions. The testing of processor units handling branches can really benefit from this.

Example: Branch Prediction Unit

Let us consider an approach for testing a Branch Target Buffer (BTB) similar to the one proposed in 4.1.4. For the example we will not consider the prediction algorithm, but just the table and its access mechanism. The test program consists in a sequence of indirect jump instructions placed in proper locations of memory, with a suitable update routine. This solution can hardly be adopted for on-line test, since it would require a test program spread along the whole code memory space. However, the suitable placement in memory of the test program can be disregarded with the MIHST approach. The program becomes only a sequence of branches and register load instructions generated by the MIHST unit. Exactly the same result is obtained from the point of view of the processor and its BTB, but the memory used by the application does not store any code and is not affected by the test.

The code shown in Figure 4.7 is written considering the test of a n-entries Branch Target Buffer Prediction Unit in a processor implementing a m bits address bus. The address in which the instructions are stored in memory (org. (pattern x) in the figure) is very important for the sake of the test coverage. However, the MIHST module is able to manage the instruction execution independently from the actual PC value in the processor. Figure 5.11 shows the MIHST-encoded instruction sequence that executes the same program.

Outer loop	Inner loops	Opcode	Operand 1	Operand 2	Operand 3
n/m	m-1	ld	r1 (inc)	00000010 (<<,rst)	-
	m	jp	r0 (inc)	-	-

Figure 5.11 Encoded MIHST-ready BTB test program.

We can notice that the length of the program is greatly reduced. However, the main advantage of the approach is that we can perform instructions located virtually at any address, whereas the system memory is not involved in the test, i.e., its content does not include any test related instruction, making this approach especially suitable for on-line testing.

5.2.5 Use of MIHST for on-line testing

A suitable methodology is defined in order to use the MIHST unit for on-line testing. We assume that in this case the test is activated via software, e.g., by the Operating System. To support test activation, the exception/interrupt features of the processor core are used; moreover, the architecture introduced in Figure 5.1

is slightly modified by adding a Flip Flop and a tri-state buffer connected to the system bus, as shown in Figure 5.12.

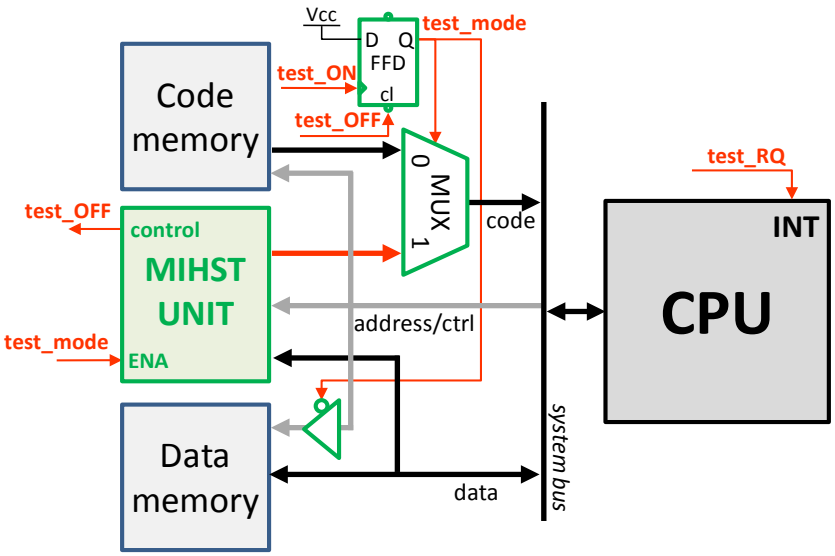

Figure 5.12 Schema of the MIHST connections for on-line usage.

When the test is activated, a proper interrupt is triggered, and a carefully crafted Interrupt Service Routine (ISR) is executed. The header and tail of such an ISR are stored in the code memory of the system, while the rest of the test instructions are fed to the processor by the MIHST module.

The proposed test flow includes the following steps:

a. The on-line test interrupt signal is activated (test_RQ signal transition).
b. The microprocessor unit saves the system context (i.e., it saves PC and flags in the stack or somewhere else, depending on the processor architecture).
c. The system control is given to the ISR, which:
    1. stores all the GPRs the test program uses,
    2. enables the MIHST module and activates the test_mode signal (pulse in test_ON signal).
d. Test_mode signal to 1 gives control to the MIHST module, which:
    3. forces the execution of a sufficient number of NOP instructions that clean the pipeline,
    4. reproduces the compacted test program,
    5. generates a new sequence of NOP instructions to align the pipeline,
    6. generates a “jump N” instruction for the CPU; N is a pre established address where a suitable jump instruction is included in the code memory; this instruction allows the ISR to return to the same point where it was when the MIHST module was enabled,

7. resets the test_mode signal and disables the MIHST module (pulse in test_OFF signal).

e. Test_mode=0 gives the control back to the ISR, which:
   8. restores the GPR contents,
   9. executes the Return From Interrupt instruction.

e. The microprocessor restores the system context prior to the interrupt and resumes its task

5.2.6 Microprocessor MIHST testing experimental results

The feasibility and effectiveness of the proposed approach have been evaluated on a system including a miniMIPS processor [**93**]. This architecture is based on 32-bit registers and addresses, and includes a 5-stages pipeline, accounting for 18,298 cells when synthesized (without multiplier) with an in-house developed library.

We implemented a MIHST unit supporting the test of the processor core. The OPCode words are 28 bit wide, whereas register OPErand words are 7 bits wide, and the immediate OPErand words are 23 bits, since the maximum seed size corresponds to 2 bytes and 7 operators for its manipulation are embedded into the MIHST unit. Test responses are observed by compressing the bus behaviour using a 72-bit MISR module.

In order to practically validate the proposed approach, we considered a couple of programs suited to test the Address Adder (4.2) and the Register File [**47**] modules following the SBST approach. The original programs were manipulated according to the method described in Section 5.2.4.a . TABLE VII and TABLE VIII show figures about the original SBST and MIHST versions of the test for these two components. The last row reports the total test time, that for the normal SBST approach includes both the time for uploading the test program code in a suitable memory within the SoC, and the test application time. The former time may be missed in the MIHST approach in case the encoded test sequence is hardwired within the unit.

TABLE VII ADDER MODULE RESULTS

	MIHST approach	**Original SBST**	**Gain %**
Fault Coverage	97.15%	97.15%	-
Application Time	700 cc	1,700 cc	58.8%
Test Program Size	97 bytes	140 bytes	30.7%
Total Test Time (including code upload)	1,422 cc	2,695 cc	51.7%

TABLE VIII REGISTER FILE MODULE RESULTS

	MIHST approach	**Original SBST**	**Gain %**
Fault Coverage	99.5%	99.5%	-
Application Time	128 cc	227 cc	43.6%
Test Program Size	89 bytes	340 bytes	73.8%
Total Test Time (including code upload)	955 cc	6,545 cc	85.4%

We also consider a third program targeting the test of the Branch Prediction Unit following the approach in section 4.1. In this case, the MIHST encoded program was developed following the guidelines introduced in section 5.2.4.b . Results are shown in TABLE IX.

TABLE IX BRANCH TARGET BUFFER RESULTS

	MIHST approach	**Original SBST**	**Gain %**
Fault Coverage	97.8%	97.8%	-
Application Time	165 cc	210 cc	21.4%
Test Program Size	17 bytes	248 bytes	93.5%
Total Test Time (including code upload)	354 cc	4,712 cc	92.5%

In all three cases, the obtained benefit is significant. First of all, it has to be noticed that the application of the original SBST and MIHST program versions return the same fault coverage for each considered module. However, the MIHST method is advantageous in terms of both test application time and code occupation. Depending on the program, the gain is higher in terms of code size (e.g., the original register file test program suits to be encoded in a few lines) or in terms of execution time (e.g., the original Adder test program includes many useless instructions from the point of view of the fault coverage, such as the loop management instructions).

In an on-line test scenario the test application time has to be minimized because the on-line test program is a process that has to compete with user processes for system resources in terms of CPU cycles. To alleviate the system's operation overhead, a test program should run in the minimum possible number of CPU clock cycles. In our case studies, the overall test execution time is reduced by approximately 50%. Only 144 additional clock cycles are spent in all the auxiliary tasks performed, because interrupts are used during on-line testing.

However, regarding on-line testing, the most remarkable advantage when using the MIHST module is the fact that the system memory remains practically unchanged. For example, the test of the BTB requires performing several jump

instructions to different and predefined memory addresses that would compromise the memory content in the pure SBST approach: this issue is automatically solved when adopting the MIHST approach. This means that thanks to the MIHST approach a higher fault coverage can be achieved, given the available resources in terms of memory area for test.

For sake of completeness, we also developed a SBST test for the whole miniMIPS processor based on the MIHST approach. TABLE X shows the achieved results and compares them to the results presented in [**87**]. Numbers regarding the Total Test Time (including code upload) are not compared as they were not included in [**87**].

TABLE X MINIMIPS OVERALL RESULTS

	MIHST approach	**[25] approach**
Fault Coverage	92.67 %	93.30 %
Application Time	4,200 cc	34,233 cc
Test Program Size	400 bytes	10,344 bytes

The results in Table IV for the MIHST approach are almost equivalent in terms of fault coverage with respect to the ones presented in [**87**]; Nevertheless, they improve greatly in terms of test program size and test program execution time. In addition to this, the MIHST approach is 100% applicable for on-line testing, while the approach in [**87**] is developed for manufacturing testing purposes, only, thus requiring full access and usage of the whole range of system memory.

We also evaluated the cost of the MIHST approach in terms of hardware for implementing the MIHST unit. TABLE XI reports the size of the various MIHST components in terms of equivalent gates when the MIHST unit for the miniMIPS processor is considered.

With respect to the overall system including two 64kbytes memory cores, the area overhead is about 1.3%. It has to be noticed that the largest parts of the MIHST unit are the embedded memory storing the encoded test program and the Test Access Mechanism interface unit; these figures can be significantly reduced if the encoded program is hardwired in a ROM space or in case an available memory core is used to store it.

TABLE XI MIHST AREA OVERHEAD

Module	Area (equivalent gates)
Control Unit	454
Embedded memories (RAM/ROM cells)	5,442 / 4,521
Operand generation module (3 instances)	940
Operand generation module	433
Instruction register	192
Result collection (MISR)	338
TAM interface unit	1,307 / 0
TOTAL (with RAM / ROM)	**9,106 / 6,878**

5.2.7 MIHST-based processor testing conclusions

The chapter proposes a hardware SBST-like method to test processor IP cores in a SoC, both in manufacturing and on-line testing application domains.

The method is based on the introduction of a special hardware module in charge of generating the instructions required for the test, which are stored in a highly encoded manner in an internal memory.

The new method provides several advantages when compared with the traditional SBST approach, while allowing to achieve at least the same fault coverage:

- It reduces the cost of the test in terms of test time: both the time for uploading the test program and the test execution time are reduced
- It eliminates the need of the code or data memory for testing purposes, thus reducing the invasiveness of the test with respect to the normal system behaviour; this is especially interesting when the method is applied to on-line testing
- It allows the execution of programs that facilitate the testing of non-functional units, as the pipeline-related logic
- It helps the owner of the processor core to better preserve the Intellectual Property, avoiding the need for distributing an explicit test program.

The cost for applying the new method is limited to the insertion of the MIHST unit, connecting it to the bus (without any change in the processor core); experimental results on the miniMIPS case show that this unit has a rather reduced size (about 1,3% of the overall system).

5.3 MIHST – An embedded memories testing strategy

System on Chip devices include an increasing number of embedded memory cores, whose test during the operational phase is often a strict requirement when safety-critical applications are considered. Possible solutions include traditional hardware BIST (based on suitable circuitry around each memory) or software BIST (based on forcing the processor to execute a proper test program). This chapter proposes a hybrid solution, in which the test is still performed by the processor, but the codes of the instructions to be executed for this purpose are generated on-the-fly by the MIHST module, which is also in charge of checking the memory behaviour. This solution can be easily adopted for the test of embedded memory modules during the operational phase; moreover, the solution is modular and does not require any modification neither in the memory cores nor in the processor. Its cost in terms of hardware requirements is limited, and the test time reduces to the one of traditional hardware BIST solutions. In addition, it solves many shortcomings that hardware and/or software BIST may suffer from. For instance, the proposed scheme: (a) provides high flexibility to deal with unexpected defects and requirements, (b) allows for at-speed testing to deal with timing related faults, (c) guarantees a high level of confidentiality as neither the memory nor the processor IP properties have to be known, (d) is compatible with standards and easily integrated into existing design flows. Experimental results gathered by implementing some representative March elements and algorithms show that the method guarantees higher defect coverage than software BIST with a hardware cost comparable with hardware BIST; moreover, test time may be reduced with respect to the former (46% with the considered example), while the required memory can also be reduced (34% with the considered example).

5.3.1 Why yet a new approach for memory testing?

Testing memory cores in SoCs is not a new topic. Solutions based on equipping cores with suitable BIST circuitry are widely adopted and represent an effective solution. However, in the last years some new issues raised, demanding new solutions.

The growing adoption of electronic systems in safety-critical applications, together with the higher sensitivity of semiconductor technologies to aging and other degrading phenomena significantly raised the importance of testing devices during the operational phase (on-line test); standards and regulations now often define the targets to be reached to achieve the desired level of dependability. When considering embedded memories [**107**], the traditional BIST solutions are sometimes not suitable to match the above constraints, e.g., because the BIST circuitry cannot be (easily) activated during the operational

phase, or because it has not been designed to support test done in this phase. Moreover, traditional BIST approaches (called here Hardware BIST, or HW BIST) are sometimes unable to test faults affecting the interconnections between the memory and the surrounding circuitry. As a result, test of embedded memories during the operational phase is sometimes performed by forcing the processor core to run suitable test programs accessing the memory and implementing in software the same sequence of read/write operations mandated by a given algorithm, often belonging to the March family. This approach (sometimes denoted as Software BIST, or SW BIST) is clearly very effective in terms of cost (since it does not require any support in terms of hardware) and flexibility (since the implemented test algorithm can be easily changed). Another advantage of Software BIST is that the test can be easily triggered whenever required.

Unfortunately, this approach also shows some drawbacks. First of all, it cannot be applied if the memory cannot be accessed by a processor. Even in the positive case, the method requires developing and validating the test code (typically written in the processor assembly language), hence requiring test knowledge to be moved from hardware designers to software developers. The task of the latter is sometimes made harder by the memory core provider, who may want to hide as many details as possible about the memory implementation in order to better preserve IP confidentiality. Moreover, Software BIST may be unable to provide the same defect coverage as Hardware BIST, especially with respect to delay faults and speed-related faults; in fact, the sequence of read and write operations is implemented through memory access instructions in the test code, which may be interleaved with other instructions intended to check the read values, to manage the loops required by the algorithm, and to prepare the address for each access (which may result to be complex when the physical organization of the memory significantly differs from the logical one). All these extra instructions prevent the fulfilment of the so-called Back-to-Back (BtB) constraint required for the detection of speed related faults [**57**][**58**][**54**][**56**]; BtB means that the sequence of read/write operations has to be performed without breaks between one access and the following. Finally, the Software BIST approach requires the test code to be stored somewhere in the application memory, thus involving some additional cost and possibly raising some issues with respect to the often strict requirements of embedded applications in terms of memory footprint.

In addition to the above mentioned confidentiality, at-speed test, and cost constraints, any suitable implementation of a memory test solution to be adopted during the operational phase should satisfy the following requirements:

- *Modularity*: solutions should be designed independently and without requiring the knowledge of the details of the different IP memory cores they should work on.

- *Programmability/Flexibility*: solutions should support the possibility of tuning the test without any major difficulty in order to deal with unexpected situations. For example, they should support the possibility of running a new test algorithm that targets some unique new faults in order to improve the defect/fault coverage.
- *Scalability*: solutions should be scalable with any design and technology, irrespective of the complexity.
- *Compatibility*: finally, solutions have to be compatible with standards, such as IEEE 1149.

This section proposes a new solution to the problem of embedded memory on-line test, while successfully targeting all the mentioned constraints and requirements. The idea is based on the insertion of the MIHST module on the bus connecting the processor to the memory. The MIHST module is completely transparent in normal mode, while in test mode is in charge of providing the processor with the flow of instruction codes implementing the algorithm for testing the memory and of checking the memory behaviour by monitoring the bus. The algorithm itself is stored in a highly encoded manner in a small memory existing within the module.

5.3.2 MIHST approach for embedded memory testing

5.3.2.a MIHST unit behaviour and usage

When exploiting the MIHST module for testing an embedded memory core during the operational phase, the same general connection scheme shown in Figure 5.1 still holds: the MIHST unit is added to the system, without introducing any change in both the processor and the memory core.

The MIHST unit stores in an internal memory a highly encoded version of the test program for the memory. When triggering the test, the processor may first execute a short piece of code in charge of saving the context, setting some registers storing for example the start address of the memory block to be tested (and its size) and switching the MIHST unit to test mode; as a consequence, the code memory is disabled and the MIHST unit starts monitoring the bus and performing two tasks:

- when it detects an instruction fetch cycle, it provides an instruction code (independently on the instruction address provided by the processor) according to the test program to be executed;
- when it detects a memory read cycle, it reads the value written on the data bus by the memory and compresses it using a MISR.

When the test is finished, the processor switches back the MIHST unit to the normal mode, reads the final value of the MISR, compares it with the expected one, and restores the context.

The MIHST unit should also support two further tasks:

- before the test is performed, and if the internal memory storing the test is a Flash, it may support the upload of a new test program in its internal memory; a standard test interface (e.g., IEEE 1149.1) may be provided for this purpose;
- at the end of the test, it should allow the processor to read the final value of the MISR; for this purpose, the MIHST unit can be accessed through the bus as a peripheral interface.

In order to support the above operations, the MIHST unit is organized according to the internal architecture reported in Figure 5.5. This organization is completely independent on the memory to be tested and only has to be customized to the system bus and processor. Therefore, it is highly re-usable.

Typical test programs for memories consist of an array of March elements, each March element implying the repetition of the same sequence of operations (writing and/or reading) throughout the whole memory. Regularity and loops are obvious characteristics of this kind of memory testing. This behaviour particularly enables the MIHST's special way of encoding, which profits from these two characteristics.

Within the MIHST unit, each processor instruction to be generated is stored in a MIHST format using four fields: one field for the operation code (OPCode) of the instruction, and three fields for the three possible operands (OPErand) of each instruction. The encoding of the OPCode embeds possible looping information, including up to two-level nested loops. The encoding of the three operands includes the possible modifications the operand may suffer in each cycle of these loops. The format used to store each MIHST encoded instruction is the shown in Figure 5.6.

5.3.3 Embedded memories MIHST testing experimental results

In this section we will show how the proposed MIHST scheme easily addresses the shortcomings discussed in the previous section with respect to the implementation of a memory test using SW BIST. Thereinafter, we will illustrate how MHIST can be used to implement memory test with complex addressing directions (such as Address Complement) and Checkerboard data-background in an efficient way while satisfying the BtB requirement. Finally, we will report about the MIHST cost and show that it is rather negligible. However, in order to

make the section better readable, the March notation and stress combinations that will be used in the examples will be first covered.

5.3.3.a March notation and stress combinations

March algorithms are the most popular memory solutions for memory testing [**107**]. An example is MATS+ defined as {⇑(w0);⇑(r0,w1); ⇓(r1,w0)}. The symbols ⇑ and ⇓ specify the Address Order (AO); they determine the way one proceeds from one address to the next address, either in an ascending order (e.g., 0,1,2,3..), or in a descending AO. MATS+ consists of 3 March Elements (MEs). The ME ⇑(r0,w1); specifies the ascending AO, and to each address a read with expected value '0' will be applied, after which a '1' will be written.

The *algorithm stress* specifies the way the algorithm is applied, and therefore influences the sequence and/or the type of the memory operations. The following algorithm stresses are of interest to this paper:

- The *Address Direction (AD)*. Memory cell arrays have a matrix organization, which means that the AO needs additional specification. Fast-row (Fr) is indicated with the AO subscript 'r' (e.g., $_{r}⇑$); it means that the row address is modified most frequently. Similarly, Fast-column (Fc) means that the column address is modified most frequently.
- The *Counting Method (CM)*. It determines the address sequence and the way one counts. The most common way is the Linear CM, denoted by the superscript 'L' of the AO (e.g., $^{L}⇑$, which specifies the address sequence: 0,1,2,3, etc.). Another CM is the Address Complement (AC) CM: as an example, for a three-bit address the $^{AC}⇑$ specifies the following address sequence: 000, **111**, 001, **110**, 010, **101**, 011, and **100**.
- The *Data Background (DB)*. It is the data pattern which actually is in the memory cell array. The DBs of interest are: solid DB (i.e., all 0s or all 1s), checkerboard DB (i.e., 0101.../1010.../0101.../1010...) row stripes DB (i.e., 0000... /1111... /0000... /1111...) and column stripes DB) (i.e. 0101... /0101... /0101... /0101...).

5.3.3.b Solutions to the critical issues

The MIHST approach offers a mechanism to effectively address the shortcomings of SW BIST revealed in Section Part I3.1.2.a . In order to better explain the concept of this mechanism and which advantages it offers, different March elements will be considered in detail and implemented first using SW BIST (to show the limitations), and thereafter using MIHST (to address them).

For the purpose of the examples, the MIPS instruction set will be considered [**93**]. Moreover, we will assume that memory address are represented on 32 bits,

and denote by $\$S_h$ and $\$S_l$, the high (i.e., most significant) and low (i.e., least significant) part (each corresponding to 16 bits) of the start address, by $\$E_h$ and $\$E_l$ the high and low part of the end address, and by \$M the size of the memory block to be tested test.

For each test case, we computed the required test time and the size of the test program; in the case of the MIHST approach, this figure corresponds to the size of the encoded test program in the memory embedded in the MIHST unit.

Loops: counting and address incrementing

Example: $\{{}^{L}_{c}\Uparrow (w0)\}$ with solid DB; i.e., linear counting method and fast-column addressing.

The implementation of the March element shown in Figure 5.13 violates the BtB requirement, thus an unrolled version can be exploited, as suggested in [**58**].

```
            ori   r18,  r0,   $M     ;Initialize Loop count
            lui   r30,  $Sh          ;Initialize START-high
            ori   r30,  r30,  $Sl    ;Initialize START-low
L1:         sw    r0,   0(r30)       ;Perform a write operation
            add   r30,  r30,  1      ;Increment MEM address
            add   r18,  r18,  -1     ;Decrement Loop count
            bne   r18,  r0,   L1     ;Branch if not equal
```

Figure 5.13 Basic SW BIST solution to loops.

```
            ori   r18,  r0,   $M/8   ;Initialize Loop count
            lui   r30,  $Sh          ;Initialize START-high
            ori   r30,  r30,  $Sl    ;Initialize START-low
L1:         sw    r0,   0(r30)       ;Perform a write operation
            sw    r0,   1(r30)       ;Perform a write operation
            …                        ;Repeat 6 times with 2-7
            add   r30,  r30,  8      ;Increment MEM address
            add   r18,  r18,  -1     ;Decrement Loop count
            bne   r18,  r0,   L1     ;Branch if not equal
```

Figure 5.14 Loop unrolled SW BIST to loops.

The partially unrolled version of Figure 5.14 matches the BtB constraint, but the test should be repeated in all the "transition" zones. With an unrolling factor of 8

(as in the example) test size increases by 87%, while the test time decreases by 60% with respect to the previous solution.

Using the MIHST approach the March element can be encoded into three MIHST instructions, as in Figure 5.15

Outer loop	Inner loops	Opcode	Operand 1	Operand 2	Operand 3
1	1	lui	r30	r0	$S_h
	1	ori	r30	r30	$S_1
	M	**sw**	**r0**	**r30**	**0 (inc)**

Figure 5.15 MIHST solution to loops.

In the MIHST version, the BtB requirement is completely fulfilled, without the need for repeating the test to cover any transition zone. Test time is further reduced with respect to the unrolled SW BIST solution by 36% and test size is reduced by 15% with respect to the basic SW BIST one (60% with respect to the unrolled version).

Result evaluation

Example: $\{{}^{L}_{c}\Uparrow (r0)\}$ with solid DB.

Test results evaluation requires the checking instruction added between memory accesses, shown underlined in Figure 5.16, thus preventing the BtB constraint to be fulfilled.

```
            ori   r18,  r0,   $M     ;Initialize Loop count
            lui   r30,  $Sh          ;Initialize START-high
            ori   r30,  r30,  $S1    ;Initialize START-low
L1:         lw    r1,   0(r30)       ;Perform a read operation
            bne   r1,   r0, Error    ;Check result
            add   r30,  r30,  1      ;Increment MEM address
            add   r18,  r18,  -1     ;Decrement Loop count
            bne   r18,  r0,   L1     ;Branch if not equal
```

Figure 5.16 SW BIST solution to result evaluation.

A possible solution is based on adopting once more loop unrolling. The unrolled version (Figure 5.17) partially solves the problem, allowing a given number (8 in the example) of BtB memory accesses.

```
      ori   r18,  r0,   $M/8  ;Initialize Loop count
      lui   r30,  $Sh        ;Initialize START-high
      ori   r30,  r30,  $Sl  ;Initialize START-low
L1:   lw    r1,   0(r30)     ;Perform a read operation
      lw    r2,   1(r30)     ;Perform a read operation
      …                      ;Repeat 6 times with {r3..r8} and 2-7
      or r9, r1, r0          ;Accumulate result 1
      or r9, r2, r9          ;Accumulate result 1
      …                      ;Repeat 6 times with {r3..r8}
      add   r30,  r30,  8    ;Increment MEM address
      add   r18,  r18,  -1   ;Decrement Loop count
      bne   r9,   r0,   Error;Check 8 results
      bne   r18,  r0,   L1   ;Branch if not equal
```

Figure 5.17 Loop unrolled SW BIST to result evaluation.

However, this implementation is limited to a maximum of 3 read operations per March element, (the unrolled program uses 8 registers per read operation, and we have 28 available registers in the processor of the example); and it also should be repeated to cover the loop transition addresses.

Using the MIHST approach the March element can be encoded into 3 MIHST instructions, as in Figure 5.18.

Outer loop	Inner loops	Opcode	Operand 1	Operand 2	Operand 3
1	1	lui	r30	r0	S_h
	1	ori	r30	r30	S_l
	M	**lw**	**r1**	**r30**	**0 (inc)**

Figure 5.18 MIHST solution to result evaluation.

In the MIHST solution results are evaluated using the MISR existing within the MIHST unit. This approach reduces the test program size by 72% and the test time by 55% with respect to the loop unrolled SW BIST. Additionally, because only 1 register per read operation is used, virtually no limit is imposed to the number of read operations per March element, as all the available registers (this number depends on the processor, in the example is 29) can be used.

Address generation

Example: $\{^{AC}\Uparrow (w0)\}$ with solid DB; i.e., Address Complement counting method.

Figure 5.19 shows the SW BIST implementation of the March element, using two different registers as destination address, in order to implement this particular counting method. This March element can be encoded into the 6 MIHST instructions of Figure 5.20.

```
      ori   r18,  r0,   $M/16 ;Initialize Loop count
      lui   r30,  $Sh        ;Initialize START-high
      ori   r30,  r30,  $Sl  ;Initialize START-low
      lui   r29,  $Eh        ;Initialize END-high
      ori   r29,  r29,  $El  ;Initialize END-low
L1:   sw    r0,   0(r30)     ;Perform a write operation
      sw    r0,   0(r29)     ;Perform a write operation
      sw    r0,   1(r30)     ;Perform a write operation
      sw    r0, -1(r29)      ;Perform a write operation
      …                      ; Repeat 6 times with {(2, -2)…(7,-7)}
      add   r30,  r30,  8    ;Increment MEM address
      add   r29,  r29, -8    ;Decrement END address
      add   r18,  r18,  -1   ;Decrement Loop count
      bne   r9,   r0,   Error;Check 8 results
      bne   r18,  r0,   L1   ;Branch if not equal
```

Figure 5.19 SW BIST solution to address generation.

Outer loop	Inner loops	Opcode	Operand 1	Operand 2	Operand 3
1	1	lui	r30	r0	$Sh
	1	ori	r30	r30	$Sl
	1	lui	r29	r0	$Eh
	1	ori	r29	r29	$El
	M/2	sw	r0	r30	0 (inc)
		sw	r0	r29	0 (dec)

Figure 5.20 MIHST solution to address generation.

This example shows that the Address Complement Counting Method (AC CM) can be efficiently implemented using the MIHST unit, thus achieving 50% size reduction and 42% test time speed up with respect to the SW BIST solution. It is to be noticed that changing the address order is straightforward.

5.3.3.c Complete March test implementation

The feasibility and effectiveness of the proposed approach is shown with the implementation of the MATS+ March algorithm [**108**], with AC CM, Fast-column addressing, checkerboard DB and on a memory with Folding = F:

$$\{{}^{AC}_{c}\Uparrow (w0); {}^{AC}_{c}\Uparrow (r0, w1); {}^{AC}_{c}\Downarrow (r1, w0)\}$$

Figure 5.21 shows the layout of an example memory block with F=4 with checkerboard DB. Counting Method notation: "a:b", a denotes Lineal whereas b denotes Address Complement.

	Col 0	Col 1	Col 2	
Row 0	0 : 0	1 : 2	2 : 4	3 : 6
Row 1	4 :8	5 : 10	6 : 12	7: 14
Row 2	8 : 15	9 : 13	10 : 11	11 :9
Row 3	12 : 7	13 : 5	14 : 3	15: 1

Figure 5.21 4-way folding memory

Following, the MIHST implementation for the three March elements of the MATS+ algorithm is detailed, starting with: $\{{}^{AC}_{c}\Uparrow (w0); \ldots\}$ in Figure 5.22.

Outer loop	Inner loops	Opcode	Operand 1	Operand 2	Operand 3
1	1	lui	r30	r0	S_h$
	1	ori	r30	r30	S_l$
	1	lui	r29	r0	E_h$
	1	ori	r29	r29	E_l$
	1	add	r1	r0	-1
	2	**sw**	**r0**	**r30**	**0 (inc2)**
		sw	**r0**	**r29**	**0 (dec2)**
		sw	**r1**	**r30**	**1 (inc2)**
		sw	**r1**	**r29**	**-1 (dec2)**
	2	**sw**	**r1**	**r30**	**4 (inc2)**
		sw	**r1**	**r29**	**-4 (dec2)**
		sw	**r0**	**r30**	**5 (inc2)**
		sw	**r0**	**r29**	**-5 (dec2)**

Figure 5.22 MIHST solution for the 1st March element of the MATS+.

To implement the second March element, data and address registers do no need to be initialized. The implementation of $\{\ldots; {}^{AC}_{c}\Uparrow (r0, w1); \ldots\}$ can be done as in Figure 5.23.

Outer loop	Inner loops	Opcode	Operand 1	Operand 2	Operand 3
1	2	lw	r2	r30	0 (inc2)
		sw	r1	r30	0 (inc2)
		lw	r2	r29	0 (dec2)
		sw	r1	r29	0 (dec2)
		lw	r2	r30	1 (inc2)
		sw	r0	r30	1 (inc2)
		lw	r2	r29	-1 (dec2)
		sw	r0	r29	-1 (dec2)
	2	lw	r2	r30	4 (inc2)
		sw	r0	r30	4 (inc2)
		lw	r2	r29	-4 (dec2)
		sw	r0	r29	-4 (dec2)
		lw	r2	r30	5 (inc2)
		sw	r1	r30	5 (inc2)
		lw	r2	r29	-5 (dec2)
		sw	r1	r29	-5 (dec2)

Figure 5.23 MIHST solution for the 2nd March element of the MATS+.

It can be noticed that both March elements have the same loop management structure; since results are evaluated by compacting them into the MISR within the MIHST unit, adding a read operation to the March element only implies adding the *load word* (lw) instruction in the proper place. This instruction loads a word from the memory to the microprocessor, hence the data will travel through the bus, allowing the MIHST to see it.

Once the memory's structure and the test's stress combinations are known, the loop management is fixed; so to invert the Address Order in order to implement the third March element, only a change in the immediate operands is needed. The suitable code snippet for $\{\ldots; {}^{AC}_{c}\Downarrow (r1, w0)\}$ is shown in Figure 5.24.

Outer loop	Inner loops	Opcode	Operand 1	Operand 2	Operand 3
1	2	lw	r2	r30	8 (inc2)
		sw	r0	r30	8 (inc2)
		lw	r2	r29	-8 (dec2)
		sw	r0	r29	-8 (dec2)
		lw	r2	r30	9 (inc2)
		sw	r1	r30	9 (inc2)
		lw	r2	r29	-9 (dec2)
		sw	r1	r29	-9 (dec2)
	2	lw	r2	r30	12 (inc2)
		sw	r1	r30	12 (inc2)
		lw	r2	r29	-12 (dec2)
		sw	r1	r29	-12 (dec2)
		lw	r2	r30	13 (inc2)
		sw	r0	r30	13 (inc2)
		lw	r2	r29	-13 (dec2)
		sw	r0	r29	-13 (dec2)

Figure 5.24 MIHST solution for the 3rd March element of the MATS+.

For the sake of comprehension the shown example works in a 4x4 toy memory. However, it is interesting to observe that this same test programs, with the same number of MIHST instructions, and hence the same test program size, can be adapted to test memories of any size and any value of F, by just adjusting the loop, initialization and immediate operands value.

TABLE XII shows the obtained results in terms of duration and test size for both SW BIST and MIHST cases.

TABLE XII TEST TIME AND PROGRAM SIZE COMPARISON FOR TESTING A SAMPLE MEMORY WITH MINIMIPS PROCESSOR

	SW BIST solution	**MIHST solution**	**Improvement**
program size	552 bytes	365 bytes	34 %
test time	158 cc	85cc	46 %

The overhead is only 5 instructions to initialize data and addresses registers (could be up to 8 for more complex relations between memory organization and

data background). This assures minimum test time, as almost all the executed instructions are memory accesses useful for testing purposes. On the other hand, the overhead for the SW BIST is 78 instructions, used to manage the loops, increment indexes and evaluate test results. Once more, the comparison results are independent on the memory size.

5.3.4 Advantages of the MIHST approach

In this Section we will discuss how the proposed method compares with the current requirements for embedded memory on-line test we mentioned in 5.3.1:

- *Confidentiality*: providing the test program for a memory core may give details about the core implementation; on the contrary, providing a MIHST unit in no way gives any detail, since the test program is encoded into its internal memory and cannot be easily transformed into a sequence of test instructions;
- *Low cost*: experimental results show that the MIHST-based solution is rather cheap in terms of hardware, since the MIHST unit is relatively small (about 3,300 equivalent gates plus the embedded memory storing the encoded test algorithm); the required amount of memory is shown to be significantly smaller than for SW BIST(34% smaller for the considered example);
- *Modularity*: in order to provide a customer with a test solution for its core, a memory core provider may only give to the customer the MIHST unit, which is a plug-and-play component fully supporting on-line test;
- *Intrusiveness*: differently than SW BIST, that requires the test code to be stored somewhere in the system application memory, the MIHST approach does not require any change in the content of this memory;
- *Ease of integration*: the adoption of the MIHST approach simply requires adding the MIHST unit to the system, without modifying in any way both the processor and the targeted memory core(s); hence, the method can be easily integrated into existing design and test flows;
- *Defect coverage*: the MIHST approach provides the same defect coverage that can be achieved with HW BIST, overcoming the BtB limitations and costs existing in SW BIST solutions;
- *Speed*: the solution based on the MIHST approach is faster than SW BIST (46% faster for the considered example) and comparable in terms of test duration with HW BIST;
- *Programmability*: the test program executed by the processor can be changed by uploading a new encoded test program into the MIHST unit, which may be equipped with an internal Flash memory accessible from the outside through a suitable interface;

- *Scalability*: the same MIHST unit can be re-used for the test of different memory cores within the same SoC, provided that they can be accessed by the processor; neither the MIHST unit nor the embedded memory size depend on the size of the memory to be tested (nor on their number).

5.3.5 MIHST-based embedded memories testing conclusions

In this chapter we propose a new approach to the test of memories embedded in SoCs performed during the operational phase.

The approach is based on the introduction of the special hardware module called MIHST unit, on the bus, without any change either in the memory core or in the processor. The MIHST unit stores in an encoded way the test program to be executed on the memory; when in test mode, it autonomously generates the suitable instructions to the processor, which executes them.

Experimental results gathered by implementing some representative March elements and algorithms show that the method can effectively overcome some limitations of the software BIST in terms of defect coverage, also reducing its requirements in terms of size of the test code and test time.

A detailed comparison of the method with respect to both HW and SW BIST characteristics show its advantages in terms of modularity, flexibility, defect coverage, cost, scalability confidentiality, and ease of integration into existing design flows.

Chapter 6
Proposed enhanced ATE – can we make it better, faster, stronger?

6.1 Diagnosis of embedded memories

This section describes the working principle and an implementation of a low-cost tester architecture supporting volume test and diagnosis of Built-in self-test (BIST)-assisted embedded memory cores.

The described tester architecture autonomously executes a diagnosis-oriented test program, adapting the stimuli at run-time, based on the collected test results. In order to effectively allow the tester architecture to interact with the devices under test with an acceptable time overhead, the approach exploits a special hardware module to manage the diagnostic process.

Embedded SRAMs equipped with diagnostic BISTs and IEEE 1500 wrappers were selected as case study; experimental results show the feasibility of the approach when having a FPGA available on the tester, and its effectiveness in terms of diagnosis time and required tester memory with respect to traditional testers executing diagnosis procedures by means of software running on the host computer.

To perform accurate diagnostic inspections is a major concern for the silicon industry. Collecting data about failure modes from every failing chip is today a recognized need, also for volume production. Tester producers and academia have proposed several architectures to cope with volume diagnosis by diminishing tester memory size requirements and introducing mechanisms for adaptively proceed along diagnostic investigation [**109**][**110**][**111**][**26**].

In the field of System-on-chip (SoC), the ability to perform volume diagnosis of embedded memories is crucial. Memory cores are currently occupying most of the chip area, therefore dominating their yield.

In theory, the adaptation process that permits to diagnose failing memory cores appears to be straightforward, since it handles decisions based on the received

outputs. The diagnostic procedure is composed of many diagnostic steps, each one depending on previously acquired chip responses. Unfortunately, to manage the adaptation process means introducing a time overhead that can be significant; moreover the test program size and data collection are aspects of the problem that have not to be underestimated.

Traditional tester architectures performing embedded memory diagnosis account on the transmission of data between tester parts, asking a host computer, or PC, to perform the adaptive calculation and decision making process through software routines, as it is shown in Figure 6.1(a). Software based processing, returning the diagnostic result as a selective composition of the received chip responses, shows two major limitations:

- the transmission of test data between host PC and tester may be slow, therefore introducing a latency within consecutive diagnostic step;
- the software routine, devoted to decide whether a new step is needed or not, and eventually to calculate parameters for the next step, may take a long time, again impacting on the latency among diagnostic steps.

In this chapter, an innovative low-cost tester architecture oriented to embedded memory diagnosis is proposed as shown in Figure 6.1(b). This architecture, equipped with an intelligent hardware diagnostic module, referred as DIA, performs the diagnostic computations that were traditionally implemented by software routines running on the host computer. Essentially, the tester is capable of autonomously managing an entire diagnostic flow, taking care of adapting each step without resourcing to any remote software, nor transferring data within tester and host PC.

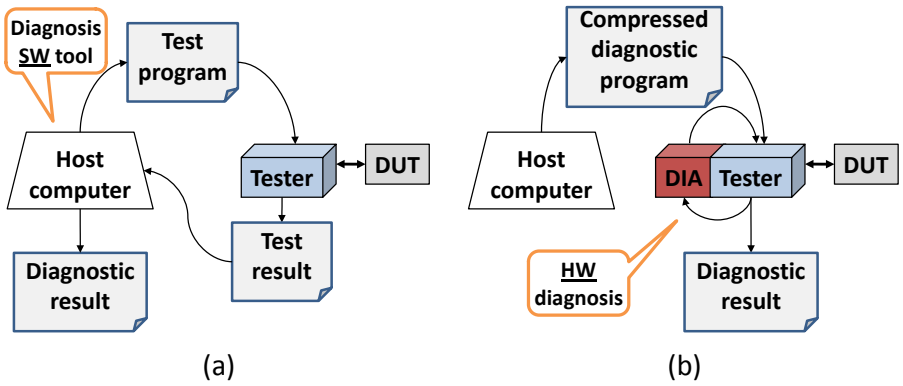

Figure 6.1 Traditional (a) and proposed (b) memory diagnosis tester architecture.

To the best of our knowledge, this is the first work proposing a methodology for adaptively supporting the diagnosis of embedded memory cores through specific

and intelligent hardware components located on tester. With respect to traditional tester strategies [**110**], [**111**] exploiting tester software capabilities, the proposed architecture speeds up the diagnosis flow, since it is able to manage an adaptive process without requesting additional software processing to the host computer.

The suitable diagnostic circuitry located on the tester is intended to be coupled with the Design for Testability (DfT) structures included on-chip such as Built-In Self-Test (BIST) modules driven through wrapper structures such as IEEE 1500 [**33**], communication protocols as *de facto* standard SPI (for Serial Peripheral Interface) and I^2C (for Inter-Integrated Circuit) [**112**], or other test pattern communication shells [**34**]. This circuitry supplies stimuli and processes the output of the Device Under Test (DUT) exploiting an ad-hoc instruction set, which enables compression of the entire set of test patterns while introducing diagnostic oriented capabilities. As a second benefit, the architecture minimizes the usage of tester storage memory. We propose a technique that try to merge many of the low-cost testing concepts towards a fast and cheap volume diagnosis process for embedded memory cores.

Experimental results collected on embedded SRAM memory included in a 90nm chip manufactured by STMicroelectronics have shown the feasibility of the approach when having a FPGA available on the tester to be coupled with the on-chip diagnostic BIST. The effectiveness of the architecture in terms of time and memory requirements during volume diagnosis was measured. To quantify the efficiency of the approach, three diagnostic flows exploiting diagnostic BIST architectures were considered, running in typical embedded SRAM failing scenarios. The section is completed by comparisons with major approaches described in the literature [**111**], [**113**].

6.1.1 Embedded memory diagnosis

Nowadays, the memory diagnosis scenario is one of the major issues for SoC producers. Three main parts shown in Figure 6.2 play roles in this scenario:

1. The BIST, a hardware module within the SoC, which applies the patterns to the memory under test;
2. The tester, which programs and launches the BIST module with appropriate test patterns.
3. The host computer, which controls the whole process, retrieves the test results and performs analysis processes in order to diagnose the device under test when necessary.

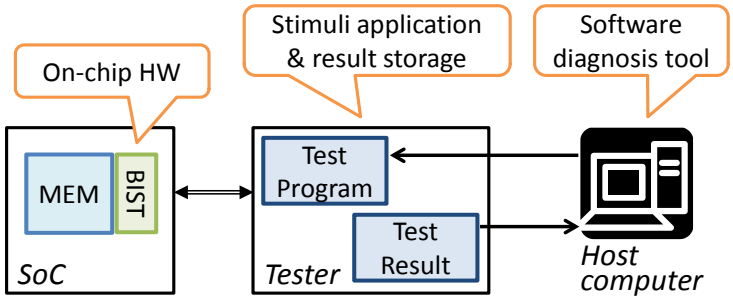

Figure 6.2 Memory diagnosis environment.

Figure 6.3 reports a basic scheme of a memory test procedure applied by a BIST module. BIST is first initialized, then the memory test is run and finally some data is read out of the chip to bin the memory core.

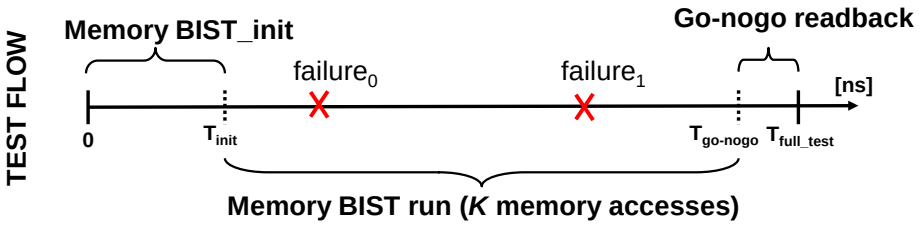

Figure 6.3 Base memory test execution with its usual phases.

In the basic BIST scheme, a go/nogo information is retrieved, simply discriminating among good and failing cores. The fault types that can be tested depend on the applied test pattern. Depending on the chosen test algorithm, different fault models are addressed, with variable test times and fault coverage percentage. For example, March SS, proposed in [**114**] is able to detect all static simple RAM faults, corresponding to six functional fault models: State Fault (SF), Transition Fault (TF), Write Disturb Faults (WDF), Read Destructive Fault (RDF), Deceptive Read Destructive Fault (DRDF), and Incorrect Read Fault (IRF).

Talking about memory diagnosis, the fault types that can be isolated still depends on the applied algorithm. In addition, it is needed a BIST architecture able to feedback many more information than the go/nogo indication. BIST architectures providing additional features to support diagnostic investigation are called Diagnostic BIST (or dBIST); they permit to store additional information other than the go/no-go feedback allowing a post-interpretation of the observed malfunctioning of the embedded memory core.

Commonly, dBIST engines as the ones described in [**41**], [**115**], [**116**] and used in [**111**] are designed to be reprogrammed for applying a slightly modified version of the base memory test procedure. In simple words, the basic test procedure is

executed several times to perform a diagnostic flow, each time customized depending on the encountered failures. The diagnostic flow is strongly based on some diagnostic facilities included in the BIST modules, such as:

- registers for storing the captured failure information, or signatures,
- features for selecting the number of memory accesses to be performed, enabling the partial test execution, and
- features for masking faults effects already considered in previous diagnostic steps.

Based on these diagnostic features, dBIST solutions usually implement one or more of the following three diagnostic flows:

Forward diagnosis [**115**]: the dBIST process is stopped anytime a fault is encountered, and this point in the test execution is internally recorded; the test execution is then restarted from the beginning by forcing the dBIST circuitry to ignore any error appearing before the test step previously identified. Figure 6.4 visualizes this concept; the diagnostic loop ends when no faults are encountered.

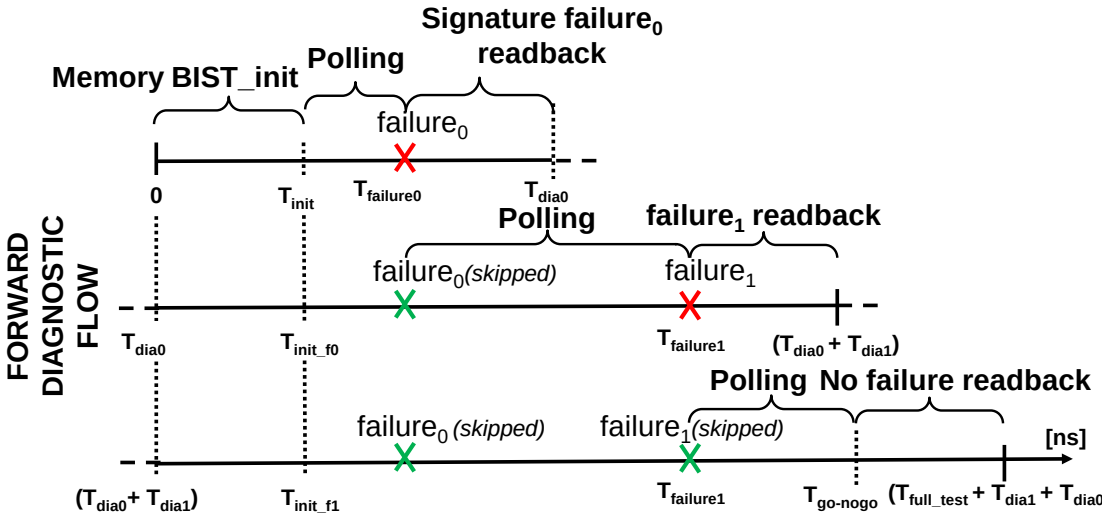

Figure 6.4 Forward diagnosis flow.

Pause and resume diagnosis [**111**], [**41**]: similar to the previous strategy, but it does not account for a dBIST restart; it simply resumes the test execution as soon as the current failure information is downloaded from the outside of the chip. Figure 6.5 illustrates this flow.

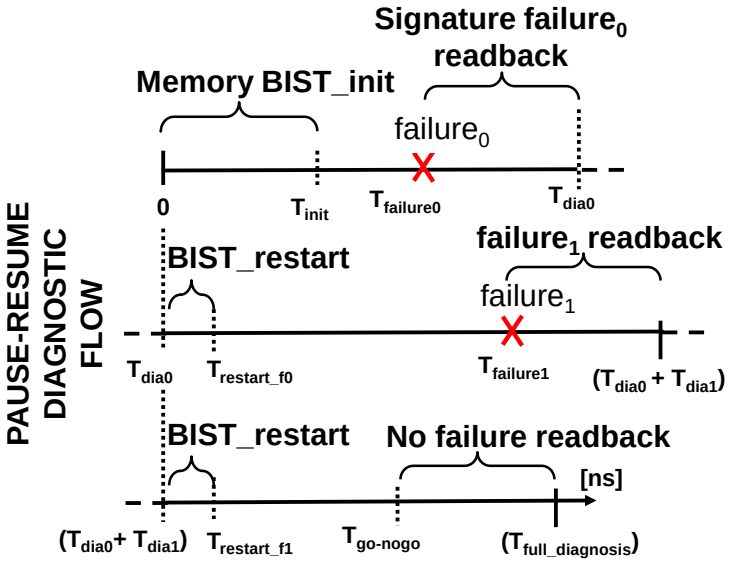

Figure 6.5 Pause and Resume flow.

Backward diagnosis [**116**]: differently from forward diagnosis, the test is run up to its end, and the last encountered fault information is stored; once it is downloaded, a new run is launched, but its end is forced to the test step preceding the last retrieved failure information. The diagnostic loop ends when no faults are encountered. Figure 6.6 shows a backward diagnostic strategy.

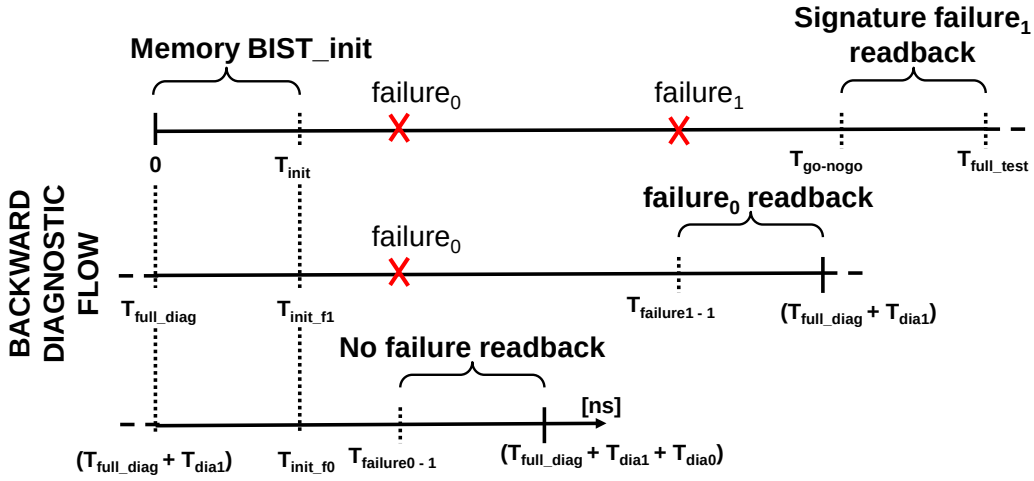

Figure 6.6 Backward diagnosis flow.

There also exist dBIST architectures able to store a set of failure information in a purposely devised buffer present on-chip aside the dBIST engine [**117**]; in these cases, a single diagnostic run may be sufficient to retrieve all the diagnostic information in case few failures are affecting the memory core. Anyway, in case of large defects, they may request for many executions in order to collect all the

information needed to build the failure bitmaps, therefore implementing one of the aforementioned diagnostic strategies. The number of diagnostic steps is dependent on both the number of failures and the size of the on-chip buffer. Solutions exploiting signature buffering introduce an additional area overhead to the chip layout that is not always acceptable.

A key point in the diagnostic process is the management of the flow, which implies the calculation of the diagnostic parameters and the generation of the pattern needed every time a new diagnostic step is required. This task is traditionally performed by the host PC that receives chip responses, performs a computation to decide whether a new diagnostic step is needed and eventually produces a suitable pattern to be applied by the tester to reprogram the dBIST.

A major issue for embedded memory diagnosis is the quantity of memory that is required on tester, both in terms of tester memory needed to store the patterns and to collect the results. In particular, the latter point may impose severe limitation to the diagnostic capabilities and additionally slow down the diagnosis time in case results have to be flushed to the host PC from the tester many times during the diagnostic process.

Still regarding the time consumed, a diagnosis process is dependent on the complexity of the computations needed to setup a new iteration; this time is negligible in case of few "spot" failures, but becomes unsustainable when considering realistic failing mechanism such as clusters, rows and other macro-defects appearing especially in new products. Furthermore, memory algorithms run today in industrial flows account for up to 40 memory accesses, thus possibly returning many failing information for each failing cell.

6.1.2 Proposed approach for embedded memories diagnosis

The tester design is thought to minimize the intervention of the host PC in the diagnostic process, thus reducing the pattern size and mitigating the time for managing the diagnosis flow. The capabilities of the proposed tester architecture progresses the state-of-the-art in the volume diagnostic tester field [**111**], providing a very efficient way to perform embedded memory volume diagnosis.

The architecture is based on following principles:

- the diagnostic flow is directly managed by the tester,
- the tester executes a diagnostic-oriented pattern set,
- the pattern is compacted and results selectively collected.

Figure 6.7 proposes a comparison between traditional and proposed approach for memory test and diagnosis. In the traditional approach (a), the test results go

back after each diagnostic step to the host PC that prepares the next diagnostic run; this flow is resulting in a slow diagnosis loop performed by software. Conversely, in the proposed approach (b), the test results are processed by DIA, an ad-hoc hardware module included in the tester directly controlling the diagnostic flow without any software intervention, thus leading to a faster diagnosis loop.

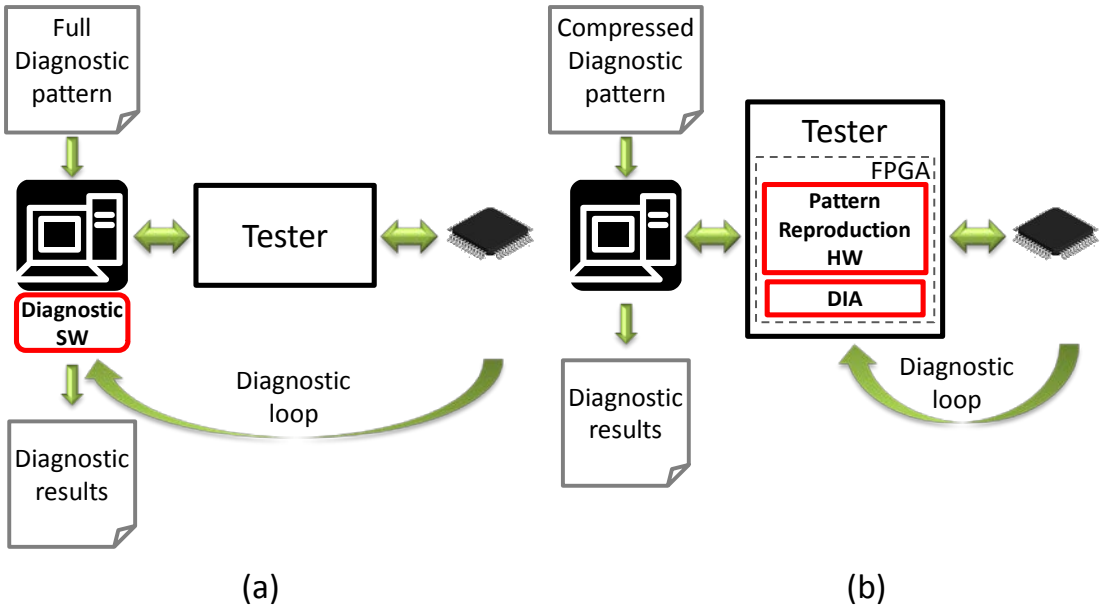

Figure 6.7 Traditional (a) and proposed (b) diagnosis approach.

The proposed tester architecture derives from several considerations and consequent efforts.

A primary consideration is related to the tester memory requirements. For making the tester able to perform the whole diagnostic process independently from the host computer, it is crucial to minimize the amount of test data it has to store. This is needed to enable the tester to proceed with the diagnostic flow without continuously requesting the host computer to intervene supplying patterns. To tackle this issue, a compression schema was devised based on the identification of repetitive test vector segments. In [**26**], the authors have already shown how an important gain could be obtained by leveraging on the identification of intensively repeated test segment parts, which is a usual scenario when test protocols are used to implement the tester to chip communication.

A second consideration is related to the design of a suitable low-cost tester platform able to cope with the compression technique. In order to reduce the cost of the equipment and maximize its flexibility, the tester design should

include an FPGA component. It is fair to state that FPGA components are usually available on commercial testers, thus this requirement is more an add-on to the traditional architecture than a tester's architecture re-spin. As detailed in the following paragraph, the FPGA is used to decompress on-the-fly the compressed pattern.

A final consideration has to be dedicated to the diagnostic requirement and to the capabilities that the tester has to own under this point of view. As a matter of fact, in a diagnostic pattern set it is known when a parameter is sent to the BIST (e.g., number of memory accesses during which failures readout have to be skipped) and when the BIST is returning test results. Therefore, these pattern parts should be clearly identified and carefully treated during the pattern compression process. From the hardware point of view, the diagnostic flow is based on some parameter calculations that in the proposed methodology are done on the tester by the DIA module, also included on the FPGA.

Figure 6.8 visualizes the conceptual flow of the proposed methodology leading to a tester able to adaptively perform embedded memory diagnosis.

The proposed low-cost diagnostic flow consists in the following steps:

1. *Test pattern analysis*: this phase produces both software resources (compressed pattern) and hardware resources (decompression logic) [**26**], and takes into consideration the diagnostic algorithm, too .
2. *Tester FPGA programming*: the resources obtained during the first phase are loaded on the tester, where the compressed pattern will be decompressed on-the-fly by a stimulation unit [**118**].
3. *Adaptive diagnosis loop*: depending on the failures affecting the embedded memory, the diagnostic module (DIA) included on the tester interacts with the stimulation unit to perform an adaptive diagnosis loop.

The final product of this complex flow is a collection of selected information that can be used to draw a so-called failure bitmap, which graphically displays the shape of the encountered failure and it is very useful for driving physical inspections.

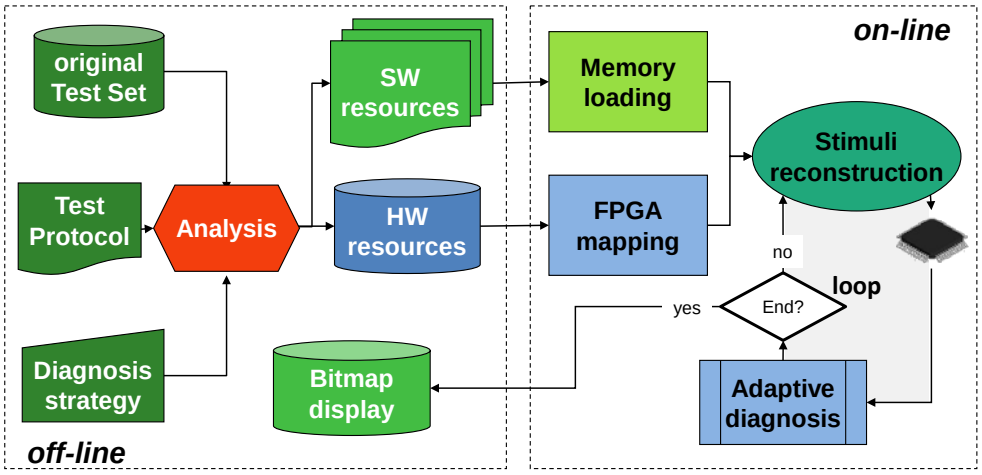

Figure 6.8 Methodology flow for embedded memory test and diagnosis.

6.1.2.a Protocol Aware Test Data Compression

The proposed compression schema is based on the identification of recurrent test set parts; recurring segments are mainly due to the Test Access Mechanism (TAM) included on-chip and to the communication protocol introduced at the SoC integration level.

The purpose of the proposed compression approach is to reduce the test data volume size by asking the tester to autonomously generate part of the test patterns that will no longer reside in the tester memory.

This phase profits from the knowledge of the employed TAM and communication protocol. Its inputs are the whole test set description, the specification of the communication protocol and the diagnosis strategy that is to be used. The analysis process returns:

- a set of hardware Finite State Machines (FSMs) corresponding to test segments identified as heavily recurrent in the original pattern,
- a modified test set description pruned from such recurrent test segments,
- the information needed to rebuild the original test set characteristics in a suitable format.

a) Recurrent test segments

Let's consider a test procedure suitable for activating self-test procedures, such as BIST execution. Independently on the implemented communication protocol, for every test data sent or read, it is possible to identify three separated phases:

1. preparation of the involved test structure(s)
2. data transfer

3. return to idle state

These three phases can be identified over the timing diagram snapshot reported in Figure 6.9, which is related to the test procedure for a SoC design driven through a TAP controller compliant with the IEEE 1149.1 standard [**13**]. Such a TAP controller is used to control the functionalities of IEEE 1500 wrappers [**33**] and its state machine is driven by five suitable top level signals.

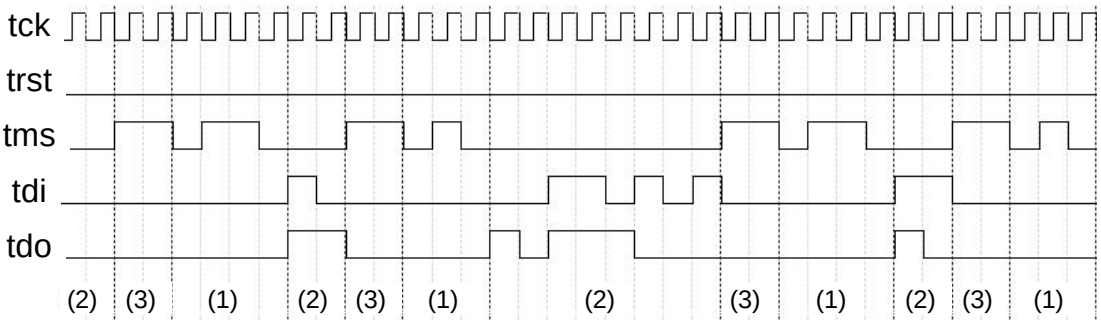

Figure 6.9 TAP access timing snapshot (zone labels are related to phases).

Depending on the content of zones labelled as (1), the TAP controller IR or the wrapper registers are addressed. Therefore, along phase (2), their content is filled up accordingly with the tdi values serially shifted in. In the example, the TAP IR is 2 bit wise (the three possible values correspond to Wrapper Instruction Register selection-write, Wrapper Data Register write, Wrapper Data Register read operations), while the wrapper register addressed is 8 bit long. During phase (3) the TAP returns to idle state.

According to these considerations, 2 types of repetitive pattern occurrences may be identified in the pattern set:

Vertical occurrences: a vertical occurrence is encountered when a timing diagram slice (during more than 1 clock cycle) is repeated many times in the overall pattern set application. Figure 6.10 shows an example; 2 vertical occurrences are observable matching with zones (1) and (3) identified in Figure 6.9.

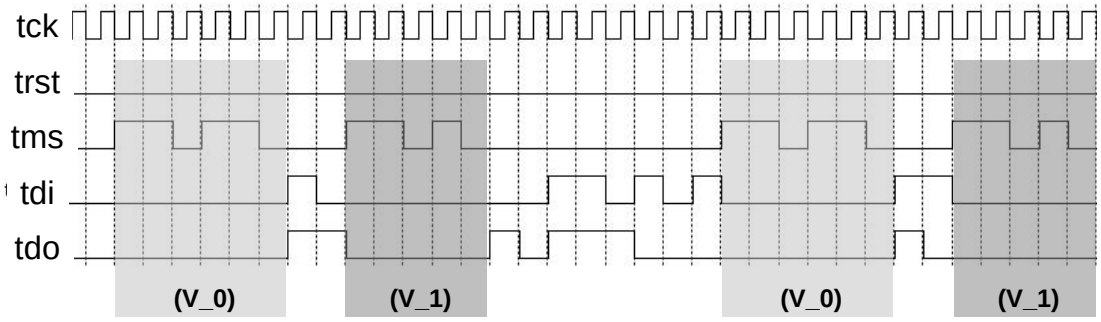

Figure 6.10 Two vertical occurrences (V_0 and V_1) identified.

Horizontal occurrences: a horizontal occurrence is encountered when a signal value is maintained stable at a certain value for more than 1 clock cycle. Horizontal occurrence identification follows the vertical one, thus it works on pattern set regions that are not vertically recurrent. An example is shown in Figure 6.11; horizontal occurrences are shaded parts with outlined border.

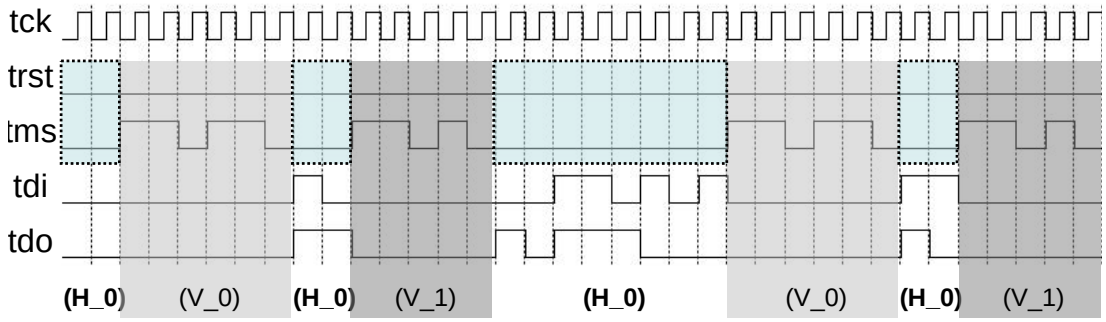

Figure 6.11 One horizontal occurrence (H_0) identified for the *trst* and *tms* signals.

By following this occurrence identification strategy, a certain number of bits (the vertical and horizontal occurrences) can be removed from the pattern set and will then be generated by hardware tester resources. The remaining of this pruning operation is therefore a *Reduced Test Set* description file (RTS file); clock cycle per clock cycle, this file stores the output bits in order. Such bits correspond to the non-coloured pattern parts in the H_0 occurrence of Figure 6.11.

A second file called *Test Segments Occurrence* file (TSO file) is necessary, storing the information required to reconstruct the original pattern set.

The following encoding has been chosen for the TSO file:

- In case the current pattern segment does not present any vertical occurrence,
  - 16 bits (2 bytes) are used to describe its characteristics
  - the MSB of the first byte is set to 0
  - the 2nd to 4th bits provide the horizontal occurrence identification number (up to 7 cases - 111 if none)
  - the remaining 12 bits store the length in clock cycle of the segment (up to 4096 clock cycles)
- In case the current pattern segment corresponds to a vertical occurrences
  - 8 bits are used to describe it in the TSO file and its value is generated by tester hardware parts
  - the MSB of this byte is set to 1

- the remaining 7 bits indicate the vertical occurrence identification number (up to 128).

In general, the reduced test data volume (RTDV) obtained by applying the illustrated method can be calculated with (6.1).

$$RTDV = 8 * (n_{TS} + n_{NRS}) + \sum_{i=0}^{n_{NRS}} l_{NRS_i} * (n_S - n_{HRS_i}) \quad (6.1)$$

Where n_{TS} is the total number of segments (either showing vertical occurrence or not), n_{NRS} is the number of segments that do not show any vertical occurrence, l_{NRSi} is the length in clock cycles of the i^{th} non-recurring segment, n_S is the number of stimulated top-level signals and n_{HRSi} is the number of signals horizontally occurring during the i^{th} non-recurring segment. In the example of Figure 6.11, the RTS file (concerning the reported test set slice) would be:

```
00001101010011111011000101110
```

The resulting TSO file is the following:

```
00000000
00000010
10000000
00000000
00000010
10000001
00000000
00001000
10000000
00000000
00000010
10000001
```

Concerning the shown example, that encompasses few clock cycles, the RTS plus TSO files size is 124 bits, while initially the test set size included 144 bits, corresponding to a test data volume reduction ratio of the 13.9%.

TSO and RTS files can be generated straightforward with a manual procedure by exploiting the test communication protocol knowledge, as in the illustrated example:

- identify the recurrent segments that will be mapped on HW,

- encode the identified repetitive sequences according to the established nomenclature,
- populate the TSO file with the generated code words,
- prune these recurrent segments from the original test pattern to produce the RTS file.

However, in order to speed the process and optimize the compression ratio, we exploit a method for automatically identifying and selecting the repetitive sequences to be pruned, presented in [**118**], based on maximal sequences search theory.

The effectiveness of this strategy depends on the analyzed patterns and it fits well with test sets showing:

- long vertical occurrences,
- few but extensively repeated horizontal occurrences.

This is the case of low-cost test strategies including BIST and SBST, which usually request many repeated initialization and result download operations.

b) Diagnostic occurrences set

In addition to the base pattern occurrence types described in section 6.1.2.a -a), some special ones are introduced, allowing the management of embedded memory diagnostic flows. These pattern segments provide the tester with some advanced capabilities, in particular enabling it to:

- Selectively capture chip responses to be used
  - to take adaptive decisions while still in a diagnostic loop
  - for further off-line failure investigation
- Understand whether the diagnostic process has ended or a new iteration is still needed
- Eventually customize some diagnostic parameters in the pattern, according to the previous output.

The special occurrences defined to this purpose are:

Variable-Data Vertical occurrence (VDV): is a sequence where all values and timings are repeated, except those related to one signal. VDV_0 in Figure 6.12 is an example, which may occur when reprogramming the BIST according with the current diagnostic step. The variable signal in VDV_0 is *tdi*; the values of the signal, that are marked as **S** in the example of Figure 6.12, correspond to the pattern parts to be substituted with the appropriate diagnostic values that may be copied from a suitable tester register at run-time.

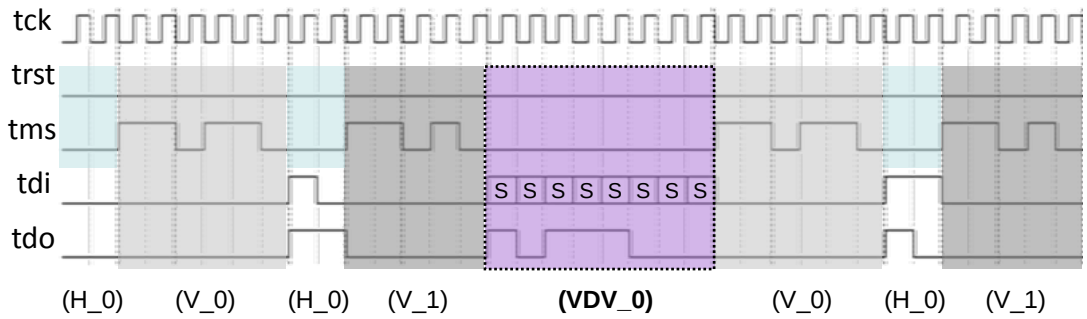

Figure 6.12 One variable data vertical occurrence (VDV_0) is identified.

Output-Recording Vertical occurrence (ORV): is a period of time during which the system has to record the output of the DUT. This type of sequence is the one used when the chip output is unknown and the data is read from the device under test. Data may be recorded in a convenient register for successive usage in the diagnostic flow. ORV_0 in Figure 6.13 is an example where the a priori unknown values of *tdo* signal, labelled as **U**, have to be read and stored in a suitable register. May be used when retrieving a faulty response.

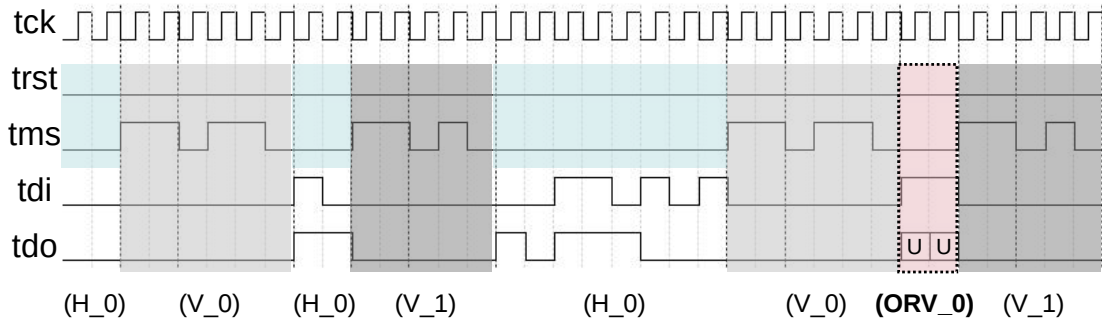

Figure 6.13 One output recording vertical (ORV_0) is identified.

Output-Polling Vertical occurrence (OPV): is defined as a timing diagram slice that at a certain point, marked as **C** in signal *tdo* in the example shown in Figure 6.14, compares the actual output of the DUT with the expected output, in order to decide whether to repeat the sequence or not. This sequence is usually used for polling the DUT status. In the proposed methodology, once an OPV is launched, it is repeated until the read value for **C** matches the expected one.

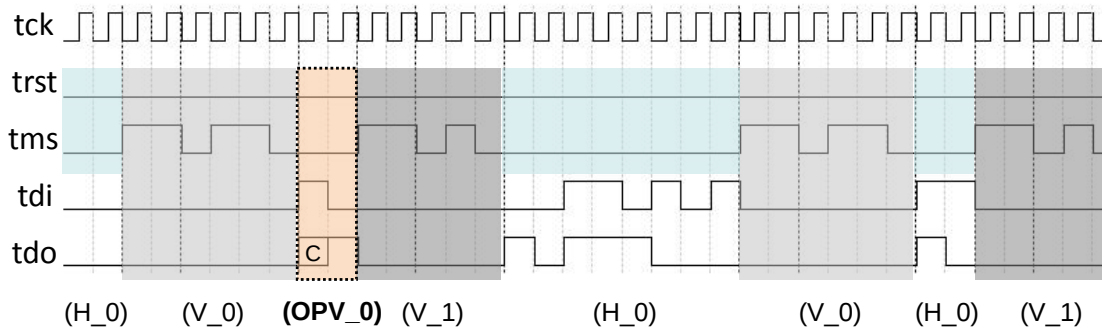

Figure 6.14 One output polling vertical (OPV_0) is identified.

Variable-Length Horizontal occurrence (VLH): they are similar to Horizontal occurrences in the sense they describe a pattern slice during which one or more signals are maintained stable at a value for a variable number of clock cycles. The difference is that the length of the VLH occurrence is not fixed but depends on the current diagnostic step and is read from a suitable tester register.

By using the denoted special purpose occurrences, it is possible to describe a single test step within a complete diagnostic run. As further described in section III.C, the proposed diagnostic-oriented tester architecture strongly exploits the special occurrences to select chip responses, allowing the tester to adaptively setup the next diagnostic step in a diagnostic flow consisting in many test steps.

6.1.2.b Low-cost tester architecture

To efficiently fetch and decode the compressed diagnosis program composed of a sequence of occurrences, a suitable low-cost tester organization was defined. This architecture is shown in Figure 6.15.

HW and SW resources are required to decompress and apply the vector set to the DUT:

- *HW resources* comprise components needed both to manage and perform decompression
  - An FPGA device including:
    - FSMs capable of autonomously reproduce the recurrent test segment parts pruned from the diagnosis program,
    - the DIA module: the circuit that enables diagnosis,
    - small and fast dual-port RAM memories.
  - A microprocessor in charge of managing the overall decompression procedure.
  - A large stand-alone RAM.
  - A DMA controller for system bus management.

  - Some communication and mass storage peripherals required to transfer data internally.
- *SW resources* include
  - The compressed diagnostic program (TSO and RTS files)
  - A suitable SW application run by the processor to manage the decompression process.

The tester is divided in 2 main components:

- The *Stimuli Controller*, is in charge of:
  - supplying the diagnostic program to the Stimuli Generator using a DMA based mechanism
  - reading the significant fail responses selected by the Stimuli Generator along the diagnostic program execution
- The *Stimuli Generator*, implemented in an FPGA, is in charge of reconstructing the test set, activating the FSMs that autonomously reproduce the recurrent test segments identified during the test set analysis. It is the most important module of the tester, since able to
  - reconstruct the stimuli previously compressed, reading the TSO and RTS files.
  - replace values in special occurrences
  - capture significant output responses
  - adaptively calculate suitable parameters for next diagnostic step.

The proposed architecture has a two-layer tester memory organization:

- secondary RAM, a large and slow general purpose memory located on the Stimuli Controller used to store the complete compressed test,
- primary RAM, a small and fast memory that receives at run-time the compressed test set to be decompressed by the FSMs based test head design, in the Stimuli Generator module.

The Memories are requested to:

- Store the TSO file
- Store the RTS file
- Store test stimuli application responses/pattern mismatches.

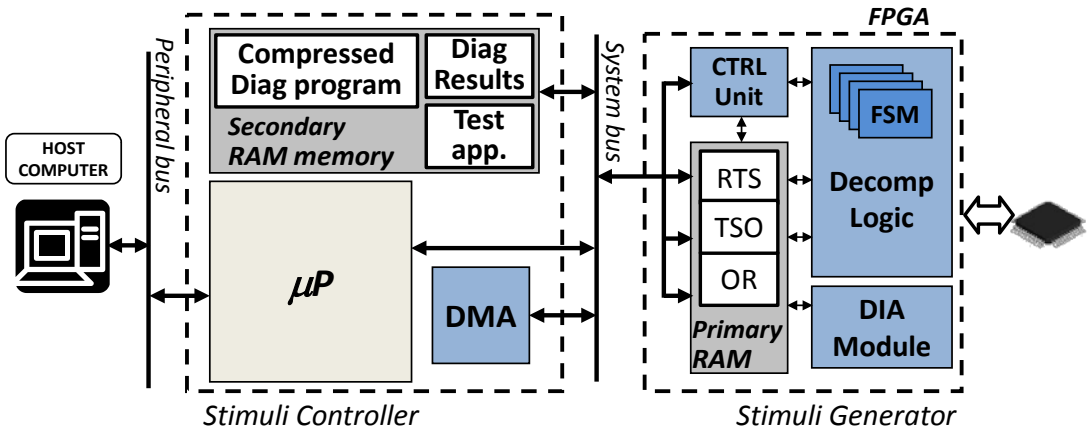

Figure 6.15 Low-cost tester architecture.

The considered tester architecture enables an effective method for data management exploiting the two-layer memory organization. In principle,

- the complete compressed pattern set is entirely stored on the secondary memory belonging to the Stimuli Controller block
- from the secondary memory the compressed information is moved block-by-block, by means of the DMA controller, to the primary RAMs, that in our schema corresponds to the FPGA dual-port RAM blocks in the Stimuli Generator block.

In terms of tester costs versus capabilities, the benefit stemming from the usage of this particular memory organization and from the compression method explained in the previous sections is twofold.

The former advantage is that a large and fast, and therefore expensive, primary memory is not required since data are transmitted block-by-block to be decompressed.

The latter benefit stems from the fact that the proposed compression /decompression schema often asks the decompression logic to run autonomously for many clock cycles without continuously requesting data. This characteristic implies that the frequency required to move the compressed data from the secondary to the primary memory may be substantially lower than the decompression frequency, thus leaving time for slow data transfer within the two memory layers.

As an example, let us consider an explanatory scenario where the primary RAM storing TSO words has only one eight bits location; if this location currently contains a word describing a vertical occurrence, an FSM is activated that reproduces a n clock cycles long sequence at the frequency f. That means the

transfer data frequency can be downed to f/n. Similarly, if a primary RAM storing RTS data has only one eight bits location, if 2 bits per fast clock cycle are used to complete an horizontal occurrence, then the requested secondary to primary memory transfer frequency is a quarter of the generation frequency.

More in general, equation (6.2) shows that the required average transfer frequency F_{trans} is function of R (compression ratio), R_v (compression ratio obtained only by vertical occurrence pruning), L_v (average length of the vertical occurrences), L_h (average length of the horizontal occurrences), B (system data bus width), F_{app} (tester to DUT frequency) and S (number of input and output signal to and from the DUT).

$$F_{trans} = \frac{F_{app}}{B}\left[S * (1 - R) + 8 * \frac{R_v}{L_v} + 16 * \frac{(1 - R_v)}{L_h}\right] \tag{6.2}$$

The obtained reduction in the required transfer frequency permits to physically separate the stimuli generator from its controller; this aspect fits the case of probe cards that actually can include FPGA cores. Our technique enables augmenting their ability in terms of stimuli application frequency while mitigating the test driver to probe card communication frequency. Moreover, it allows many stimuli generator blocks to be managed by a single controller.

The diagnostic abilities of the tester reside mostly in the Stimuli Generator Unit. It contains the Finite State Machines that have been identified during the analysis phase and the DIA module, along with some memory blocks.

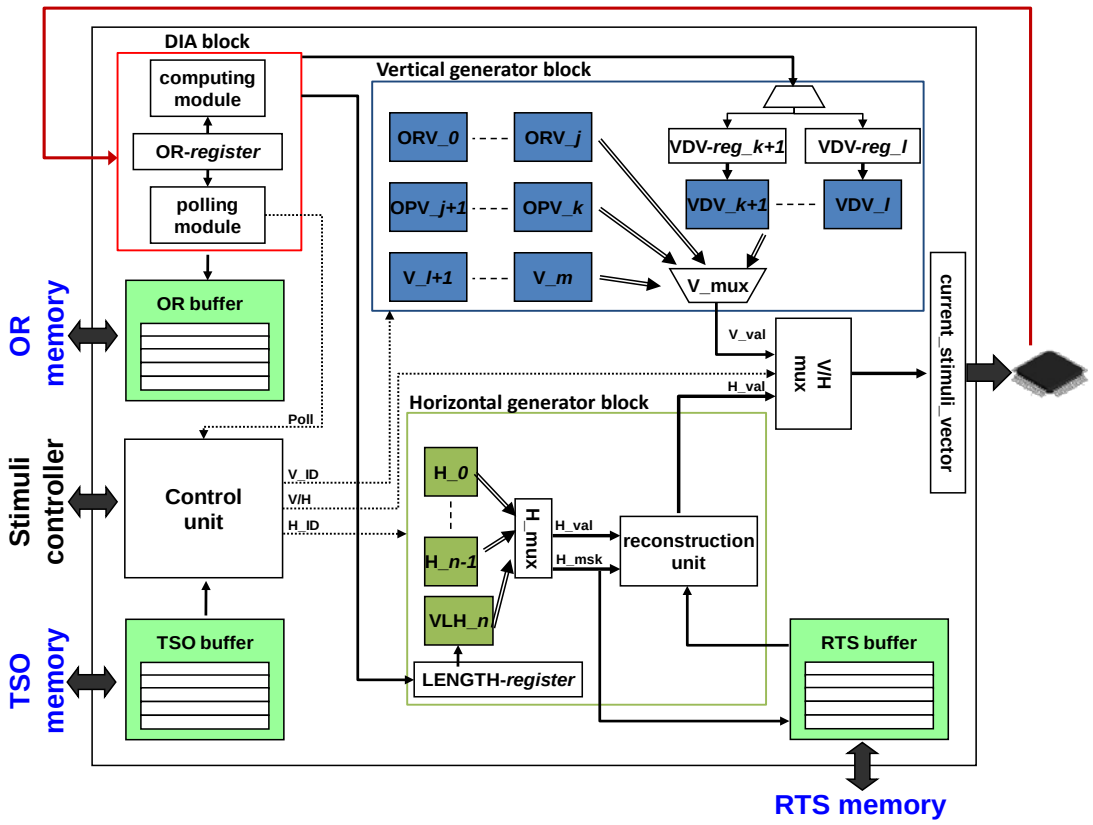

Figure 6.16 Stimuli Generator conceptual view.

This block-based design implies that when moving from a diagnosis program to another, only the Finite State Machines should be redesigned, in accordance with the identified occurrences; while the rest of the Stimuli Generator hardware remains unchanged.

Five main blocks can be found in the Stimuli Generator unit, as shown in Figure 6.16:

Control unit: it manages the whole process; being in charge of both communicating with the Stimuli Controller and managing the test stimuli reconstruction/application and the diagnosis procedure. The first task comprises receiving the compressed diagnostic program, sending the results and exchanging test procedure management signals, e.g.: test launch, status polling and conclusion acknowledgement. The latter aspect consists in reading instructions from the TSO buffer and translating them into signals used as selection signal of suitable multiplexers:

- V_ID signal: identifies the specific vertical occurrence to be reproduced,
- H_ID signal: identifies the specific horizontal occurrence to be reproduced,

- V/H signal: this signal enables the application of patterns deriving from FSM reproducing horizontal occurrences or vertical ones.

In order to manage the diagnosis procedure it receives feedback from the DIA block and takes the appropriate decisions about

- repeating or not a polling cycle
- executing a new diagnostic step
- ending the diagnostic loop.

Vertical generator block: it outputs the value reproduced by the selected Finite State Machine (FSM), according to the current V_ID read in the TSO buffer by the Control Unit; when a VDV is selected, the FSM data are combined with the content of a suitable register called *VDV-register*.

Horizontal generator block: it outputs the FSM selected by H_ID signal; merging it with data from the RTS buffer, by means of the reconstruction unit. A standard horizontal occurrence is reproduced during the period of time specified in the TSO word read by the Control Unit; when a VLH occurrence is selected, the length of the horizontal sequence is determined by the content of the *LENGTH-register*.

DIA block: it compares, at every clock cycle, the obtained output with the expected one, giving appropriate feedback to the Control Unit. For diagnosis purposes, it performs the following operations:

- using the polling module, it checks the test status by observing the response of OPV occurrences, and contrasting it with the end-loop condition (stored in the FSM of the same OPV), asking the control unit to repeat the polling sequence or to continue the pattern application, accordingly,
- it records the relevant outputs (such as the read-back signatures) relying on the execution of ORV occurrences and exploiting the OR-register,
- in the computing module, it implements the functions needed for the diagnosis procedure implementation. f computes the values to be stored in the VDV-registers and g computes the length values to be fed to the *LENGTH-register*. Both values will then be used by VDV and VLH occurrences respectively. Functions f and g are usually simple increment, decrement or shift functions, depending on the BIST unit of the DUT and the diagnostic strategy used.

Memory blocks: there are three memory blocks in the design working as circular buffers: RTS, TSO, and OR.

TSO and RTS buffers, managed by two independent managements units, send data to the corresponding blocks when requested (Horizontal generator block and Control unit, respectively); and when the buffers are half empty they request and receive data from the corresponding secondary memory in the Stimuli Controller where the full content of the TSO and RTS files is stored.

The OR buffer, receives data from the DIA block at run-time with the proper test results; and these data is transmitted to the Stimuli controller secondary memory when the OR buffer is half full.

6.1.2.c Diagnostic procedure application

Using the proposed architecture, the stimuli generator is able to implement any of the diagnostic flows described in section 6.1.1. The described components allow the tester to perform these diagnostic processes autonomously, selectively collecting results and without any need for external control or data transfer. Towards the application of diagnostic flows, a crucial role is played by three groups of diagnostic registers, already cited in the previous paragraphs:

- *VDV-registers*, supporting the completion of **S** in VDV occurrences
- *LENGTH-register*, controlling the execution length of VLH occurrence
- *OR-register*, supporting the capture of the **U** value in ORV segments.

The implementation of the Backward (Figure 6.17 and Figure 6.18), Forward (Figure 6.19 and Figure 6.20) and a Pause and Resume (Figure 6.21 and Figure 6.22) diagnosis flow diagrams and high level description is graphically shown.

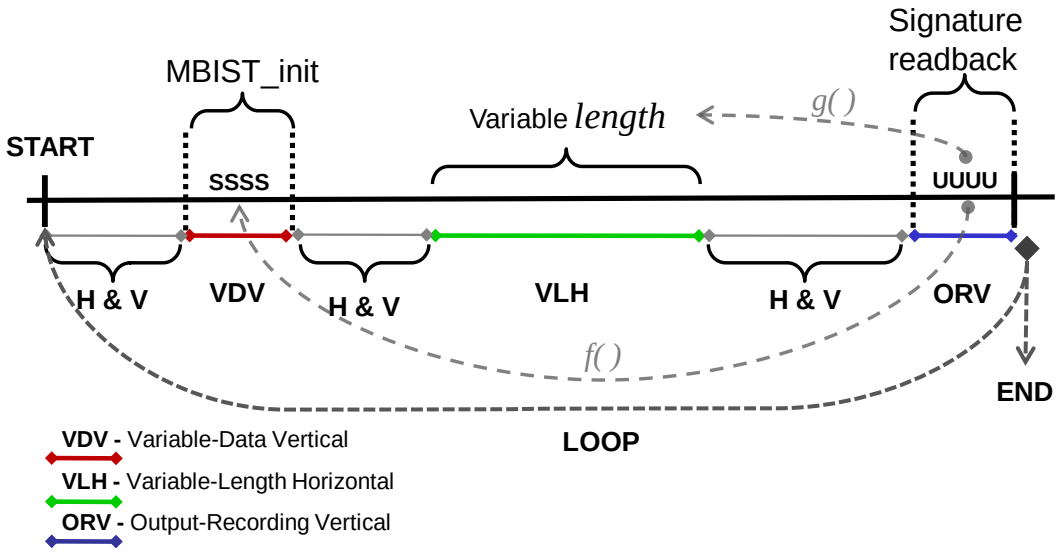

Figure 6.17 Backward diagnosis flow diagram.

```
Backward_diagnosis(FULL-LENGTH,FULL-INIT){
1. START:       LENGTH-register ← FULL-LENGTH;
2.              VDV-registeri ← FULL-INITi;
3. GO:          read-from-TSO();
4.              case(Vi or Hi) {
5.                   reproduce;
6.                   goto GO;}
7.              case VDVi  {
8.                   reproduce(@ SSSS=VDV-registeri);
9.                   goto GO;}
10.             case VLH   {
11.                  reproduce(@ length=LENGTH-register);
12.                  goto GO;}
13.             case ORVi{
14.                  OR-register ← UUUU;
15.                  VDV-registeri ← f(OR-register) ;
16.                  LENGTH-register ← g(OR-register);
17.                  if (LENGTH-register != 0)
18.                       goto GO;}
19. END: }
```

Figure 6.18 Backward diagnosis iterative process high level description.

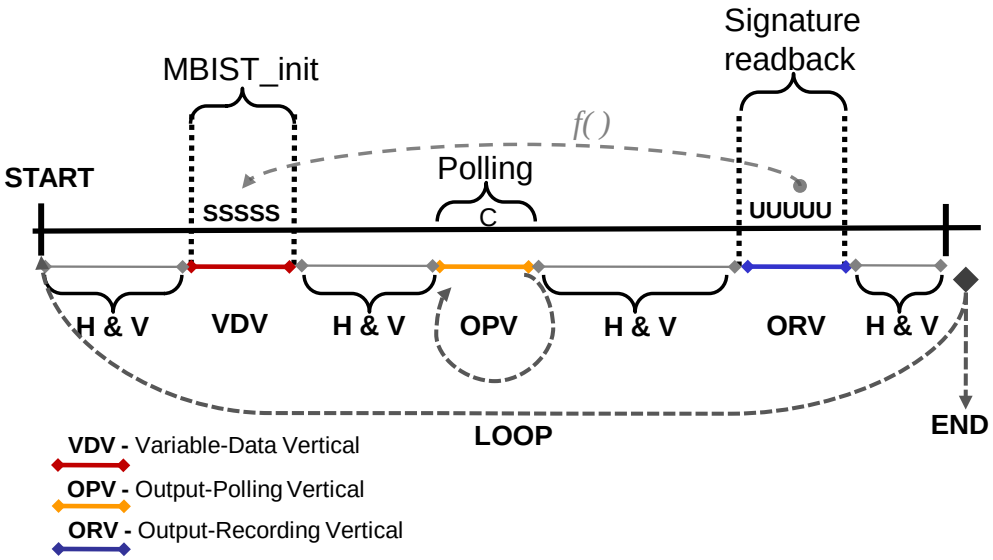

Figure 6.19 Forward diagnosis flow diagram.

```
Forward_diagnosis(FULL-INIT){
1. START:       VDV-registeri ← FULL-INITi;
2. GO:          read-from-TSO();
3.              case(Vi or Hi) {
4.                    reproduce;
5.                    goto GO;}
6.              case VDVi {
7.                    reproduce(@ SSSS=VDV-registeri);
8.                    goto GO;}
9.              case OPV {
10. POLL:             reproduce;
11.                   if (C != 1)
12.                         goto POLL;
13.                   else
14.                         goto GO;}
15.             case ORVi{
16.                   OR-register ← UUUU;
17.                   VDV-registeri ← f(OR-register);
18.                   if (OR-register!=END)
19.                         goto GO;}
20. END:}
```

Figure 6.20 Forward diagnosis iterative process high level description.

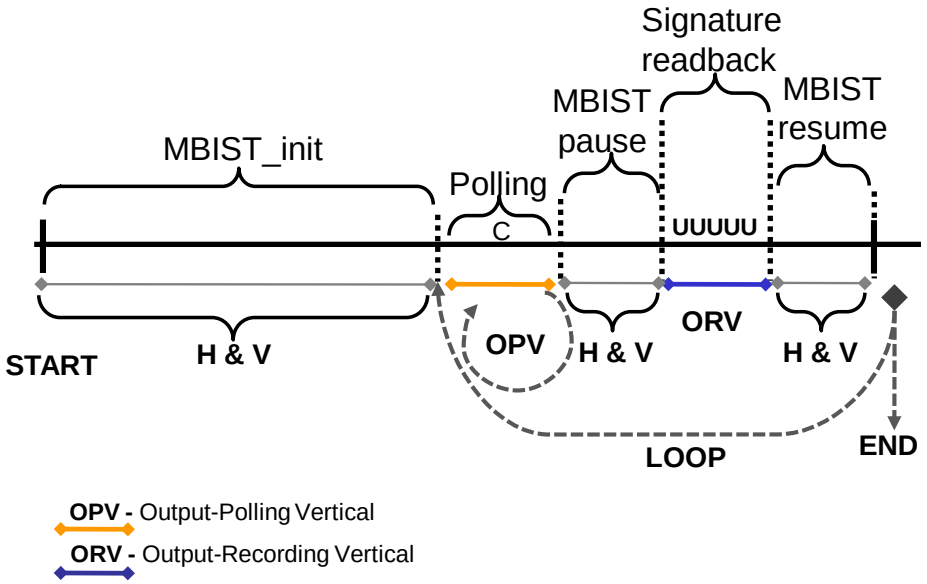

Figure 6.21 Pause and Resume diagnosis flow diagram.

```
Pause_and_Resume_diagnosis(){
1. START:
2. GO:          read-from-TSO();
3.              case(Vi or Hi) {
4.                    reproduce;
5.                    goto GO;}
6.                    case OPV {
7. POLL:              reproduce;
8.                    if (C != 1)
9.                          goto POLL;
10.                   else
11.                         goto GO;}
12.             case ORVi{
13.                   OR-register ← UUUU;
14.                   if (OR-register!=FINISHED)
15.                         goto GO;}
16. END:}
```

Figure 6.22 Pause and Resume diagnosis process high level description.

In the reported diagnostic loop description, functions f and g are executed by the Computing Module in the DIA block.

6.1.3 Experimental results for the embedded memories diagnosis

Several aspects were considered in the implementation and evaluation of the proposed architecture. The experimental results reported were obtained on a 90nm chip manufactured by STMicroelectronics, whose diagnostic features are detailed in section 6.1.3.a . The prototype tester implemented is described in 6.1.3.b , where the FPGA occupation and working frequency are quantified. Following, sections 6.1.3.c 6.1.3.d and 6.1.3.d illustrate the advantages obtained in terms of time and memory requirements.

For the sake of measuring the introduced benefit, a set of comparisons with other state-of-the-art approaches is provided. In particular, we provide comparisons for diagnostic time [**111**] and tester data volume reduction [**113**].

6.1.3.a Diagnostic BIST scenario

The proposed methodology was evaluated on a 90nm SoC including embedded memory cores equipped with a programmable diagnostic Built-In Self-Test (dBIST) circuitry whose architecture is detailed in [**119**] and graphically shown in Figure 6.23; in addition to the memory core, the SoC includes other cores which exceed the scope of this paper, and are not graphically reported.

The dBIST was used to diagnose a medium size, 39KB embedded SRAM memory (13 address bits and 39 bit word parallelism) with a scrambled organization implementing a multiplexing parameter equal to 16, running a 36n March algorithm.

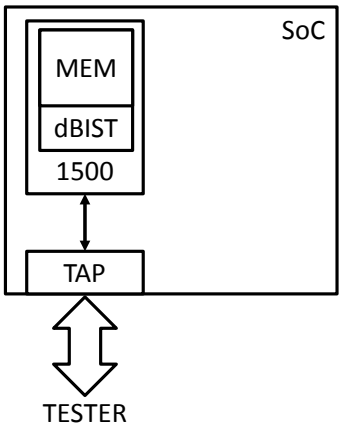

Figure 6.23 Embedded memory test infrastructure in the case study SoC.

The diagnostic BIST microcode upload, execution monitoring and results retrieval are managed through an IEEE 1500 wrapper driven by an IEEE 1149.1 (JTAG) TAP interface supporting a communication protocol based on diagnostic

commands [**119**]. The usage of this low-cost DfT interface permits to independently consider the test application frequency and the test management frequency. In particular, the test application can be supported by high-speed free-running clock sources (such as PLL) located on-chip, while the tester to BIST communication frequency may be lower. This feature allows a less complex tester architecture and subsequent cost reduction. In the experimental scenario, the dBIST was supplied by a 200MHz free-running clock generated by a PLL.

The diagnostic BIST can be programmed to run March tests and to implement 2 types of diagnostic procedures: forward and backward approach.

The forward approach that can be implemented by the dBIST bases on the following features

- The dBIST engine can be programmed by setting a (13+6) 19 bits register with the number of memory test steps that have not to be monitored for failures
- It is able to poll the memory test state by reading a 2 (1+1) bits register; the first bit indicates that the test is paused, the second that a fail information is stored in a 58 (19+39) bits register
- In case of fail detected, the dBIST engine can be asked to return the content of such the 58 bits register; including the failing step (19 bits) and word mask (39 bits). By elaborating the gathered failing step value, the next diagnostic run can be setup.

Conversely, the backward strategy applied using the dBIST is based on the following features

- The dBIST engine can be programmed by setting a 19 bits register to the number of memory test steps to be executed
- It returns the last failing step for every test run which is stored in a (19+39) 58 bits register including again the failing step and word mask. By elaborating this information, the next diagnostic step is setup if needed.

6.1.3.b Low-cost tester implementation

The FPGA-based low-cost tester was prototyped using a Digilent XUP development board [**120**]. This commercial product is equipped with two PowerPC processors connected to a 256MB std-alone DRAM (secondary RAM); the board also includes a Virtex II family FPGA containing about 30k logic cells and 2 Mb of RAM blocks (primary RAM) [**121**]. The system bus available for connecting processors, DRAM and FPGA resources permits transferring 64 data bits per system clock cycle. The DMA controller was included in the system as a

soft-core mapped in the FPGA. We used the Linux kernel 2.6 as Operative System (OS) and wrote an assembly procedure to manage the compressed pattern and result transfer within the secondary RAM and the BRAM blocks of the FPGA. The length of this program is about 200 code lines.

Since the OS is directly running on the board, the diagnostic process can be managed and monitored from the host PC through an Ethernet remote connection.

The tester finally was connected to a daughter board hosting a single SoC. This board provides the chip with correct levels of power supply and was used for chip validation. Figure 6.24 shows the experimental setup.

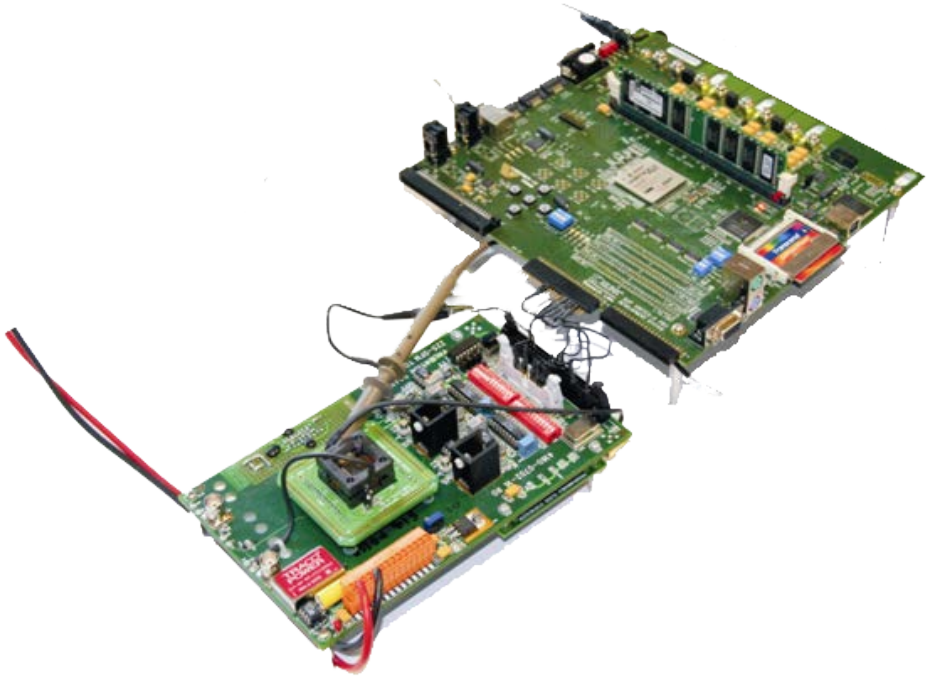

Figure 6.24 Experimental setup of the tester.

The hardware needed to implement the tester architecture was synthesized using the software suite Xilinx ISE v10.1and mapped on the FPGA of the development board. Its final occupation is 978 4 inputs LUTs, 442 FFs and 3 BRAM block of 18 Kb each one. Detailed values are reported in TABLE XIII.

TABLE XIII TESTER ARCHITECTURE FPGA OCCUPATION

	LUTs	**FFs**	**BRAMs**
Control Unit	78	44	-
Sequences Generator Block	492	165	-
DIA Block	355	208	-
Memory block	53	25	3
Total	**978**	**442**	**3**

Some of the tester components meant to be embedded on FPGA does not require any redesign when changing the analyzed pattern (e.g., the buffer managers and the Stimuli Generator control unit). Therefore, fix parts are designed once in VHDL and integrated with the FSM VHDL descriptions resulting from the pattern analysis phase and the suitable computation unit.

It has to be noticed that all the blocks implementing the proposed architecture, including the DIA module, are located on tester. So, even if the hardware overhead of the methodology is not negligible, it has reduced costs and it is less intrusive than BISD methodologies that imply on-chip logging and/or compressing of the test outputs [**117**]. In these cases, on-chip memory and additional hardware are needed to log and process the test responses, therefore introducing an additional silicon cost, not always sustainable by manufacturers. In the presented approach, the hardware does not affect the chip area nor influences in its design; it is incorporated into the tester, by means of an FPGA.

The design can work at a frequency of 220 MHz; this frequency shall vary according to the operation implemented by the computing module within the DIA block, which depends on the test performed and the diagnostic method used. In the illustrated case the only calculation to be done is an increment or a decrement of a value, indeed the generation frequency is quite high considering that this is not a specialized, performance-oriented FPGA.

Anyway, mainly due to imperfect wire connection within developed tester and to the XUP board intrinsic limitations, the FPGA to DUT communication frequency was finally limited to 50MHz. With this communication frequency and in the specific case of JTAG driving IEEE 1500, the average secondary to primary RAM transfer frequency required is 4.4 MHz (19.4 MHz for a 220MHz communication frequency). With this frequency values, a 36n memory test execution takes about 1ms.

6.1.3.c Volume diagnosis time gain

To underline the benefits in terms of diagnostic time saved, the results coming from the proposed tester architectures are compared with those produced by a version of the tester without hardware diagnostic abilities. In this alternative version, the stimuli generator simply logs failing bits and sends them to the stimuli controller, which runs a SW application understanding relevant pattern parts and setting up a new test step after parameter calculation (every diagnostic operation is performed through software, only). This traditional approach is described on [**111**] and, basing on our experience, we believe that this is the most frequent solution implemented by commercial testers, where diagnostic abilities are implemented by software routines.

In [**111**] it is also proposed a solution that mitigates the negative effect of software calculation over the times for test and diagnosis; a comparison with this technique is provided in the next subsection considering the case of pause-resume diagnostic strategy.

This point forward, the proposed architecture will be referred to as the HW-diagnostic tester, while the other will be the SW-diagnostic tester.

To compare the abilities of HW- and SW-diagnostic tester capabilities, we selected two typical failing scenarios among those currently observed in embedded SRAM. These examples quantify the cost for the diagnosis of a single failing SRAM core, considering Backward, Forward and Pause and Resume diagnostic strategies.

Following this analysis and to better support the evaluation of the proposed method, the benefit over a volume of a lot (100 wafers) is also reported and discussed.

1. Two typical faulty scenarios and their cost for diagnosis

Usually, memory failing mechanisms lead to few failing shape categories [**27**]. In this section two typical failing scenarios are considered and the cost of diagnosis is calculated.

The first faulty scenario includes a couple of small defects. On the contrary, the second is considering a failing scenario that includes line and row defects.

Each one of the failing cells individuated in the failure bitmaps of Figure 6.25 and Figure 6.26, may correspond to several information readouts done during different march elements, depending on the encountered fault type. In both the scenarios, the related diagnosis length was calculated by separately considering test (high = 200MHz) and management (low = 50MHz) clock frequencies.

The overall computational time reported in the following accounts for two contributions:

- the pure diagnosis time (T_{pure_dia}), which is the diagnosis time calculated without considering any latency time introduced by the tester when moving from a diagnostic step to another
- the tester latency time ($T_{latency\_dia}$), which is the time required by the tester to compute the next step calculations and setup the sequence of test runs.

Scenario 1 (Figure 6.25)

- One 2*2 cluster (four stuck-at-0 fault – SA0)
- One spot (one stuck-at in the address decoder fault – AF)

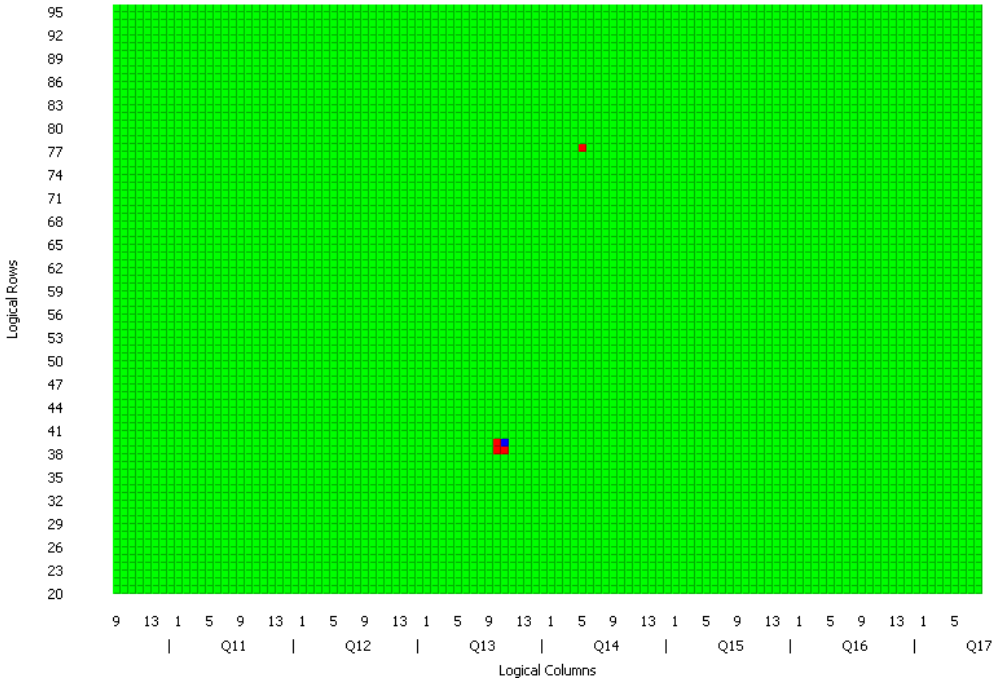

Figure 6.25 Faulty scenario 1: cluster + spot fail.

The memory diagnosis procedure is composed of sixty one test repetitions. TABLE XIV shows T_{pure_dia} required to apply the diagnostic flow. Each SA0 fault is detected fifteen times, while the AF is detected only once.

TABLE XIV FAULTY SCENARIO 1
DIAGNOSIS CLOCK CYCLE COUNT AND TIME

Forward/ Backward diagnosis strategy	
High-speed clock cycles (#)	~10M
Low-speed clock cycles (#)	~4K
T_{pure_dia}	~50ms

The pure cost of diagnosis shown in TABLE XIV is relatively high, even in a faulty scenario not accounting for a large number of failing cells. The major cost is constituted by the memory algorithm repetitive execution.

It should be also noticed that backward and forward diagnosis strategies account for almost the same T_{pure_dia}; actually, the forward strategy is slightly slower due to the polling operation that may introduce a short latency. Anyway, backward and forward times are not significantly different.

*Scenario 2 (*Figure 6.26*)*

- A partial failing column (327 stuck-at-1 fault – SA1)
- A partial failing row (31 stuck-at-0 fault – SA0)

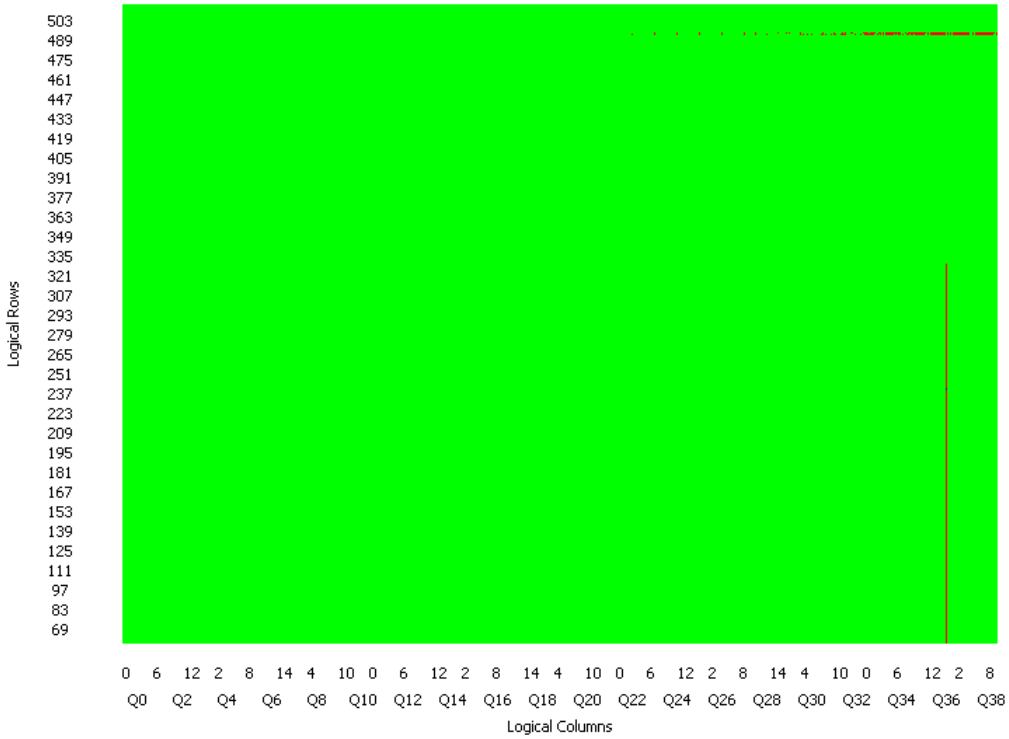

Figure 6.26 Faulty scenario 2: partial column + partial jeopardized row.

In this case, the memory test is returning 5,370 failing steps. TABLE XV shows the number of clock cycles required to apply the diagnostic flow; figures include test steps execution (high frequency) and BIST initialization/read result procedures (low frequency).

TABLE XV FAULTY SCENARIO 2
DIAGNOSIS CLOCK CYCLE COUNT AND TIME

Forward/ Backward diagnosis strategy	
High-speed clock cycles (#)	~845M
Low-speed clock cycles (#)	~375K
T_{pure_dia}	~4.2s

With respect to the former case study, the number of memory test repetitions requested in this scenario makes the diagnosis time much longer.

Let's now consider the $T_{latency\_dia}$ times introduced by the SW-diagnosis tester and by the proposed HW-diagnosis tester. To measure this time value, we considered equation (6.3).

$$T_{latency\_dia} = T_{compute} + T_{rd/wr\_dia} \quad (6.3)$$

being:

- $T_{compute}$: time needed to computer the diagnosis parameter for the setup of the next diagnostic step
- T_{rd/wr_dia}: time overhead introduced when moving failing data to the tester memory and back to the dBIST, in order to setup a new diagnostic step.

For the proposed HW-diagnosis tester, $T_{compute}$ is equal to two clock cycles, and there is no need to transmit any diagnostic data to the secondary memory to perform diagnostic computation.

On the contrary, in the SW-diagnosis tester architecture, $T_{compute}$ was measured equal to 0.1ms (computation performed by an optimized routine written in assembly language) and a T_{rd/wr_dia} was measured as 0.05ms.

TABLE XVI shows the $T_{latency\_dia}$ for the two illustrated scenarios including the overhead percentage with respect to T_{pure_dia}.

TABLE XVI TIME OVERHEAD COMPARISON

Backward diagnosis strategy			
		SW-diagnosis	***HW-diagnosis***
	T_{pure_dia}	~50ms	
Scenario 1	$T_{latency\_dia}$	~9ms	2.44 µs
	Overhead	18%	0.0049%
	T_{pure_dia}	~4.2s	
Scenario 2	$T_{latency\_dia}$	~805ms	215 µs
	Overhead	19.2%	0.0051%

In both cases, the time overhead introduced by the SW-diagnosis tester is quite significant; it reaches almost 20%. Conversely, the proposed HW-diagnosis tester is introducing an overhead that can be considered negligible with respect to the pure diagnosis time.

In case the Pause and Resume diagnostic flow is adopted, an optimized software solution may be used, as described in [**111**]. The methodology in [**111**] tries to minimize the time overhead introduced by diagnostic operations, as pursued by our technique. For embedded memory diagnosis, it consists in a preliminary complete run of the memory test, to record every time point where a failure appears; this collection of fail points is then used to generate a diagnostic test program that is fed to the DUT in a second run, this time also reading back the whole diagnostic information needed to build a failure bitmap.

TABLE XVII COMPLETE TIME COMPARISON

Pause and Resume diagnostic strategy			
		SW-diagnosis [3]	***HW-diagnosis***
Scenario 1	T_{pure_dia}	3.03 ms	1.56 ms
	$T_{latency\_dia}$	50 µs	2 µs
	Total	3.08 ms	1.56 ms
Scenario 2	T_{pure_dia}	10.46 ms	8.99 ms
	$T_{latency\_dia}$	2.61 ms	215 µs
	Total	13.07 ms	9.20 ms

Table V compares the time for diagnosis in both cases. It should be noticed that, in case of few faults, the SW-diagnostic tester takes longer than the HW-diagnostic one, mainly due to the fact that the former applies two times the memory test; for this reason, the T_{pure_dia} is different for the considered cases. For large amount of fails, the SW-diagnostic is still slower than the HW-diagnostic tester, this time because of the generation and transfer of the diagnostic program, leading to a substantial overhead.

2. Volume diagnosis benefit

To support the discussion about overhead mitigation that is obtained by using a HW based diagnostic mechanism, a projection of the results was applied to an entire lot, composed of 100 wafers. Considering wafers including 1,000 chips, a fail rate of 10 chips per wafer (showing corrupted memories) and an average of 25,000 failing steps per wafer, the pure test and diagnosis time for each single wafer may be in the order of 25 seconds.

In case of using a SW-diagnosis tester, an additional overhead due to the latency of SW in managing the backward diagnostic preparation has to be considered. In the volume scenario, such an overhead can be quantified in about 4 seconds, corresponding to about the 15% of the overall processing time. Considering a whole lot consisting in a set of 100 wafers whose pure test and diagnostic time is about 40 minutes, the overhead using the SW-diagnosis tester is about 7 minutes.

If using the proposed HW-diagnosis tester, this overhead becomes negligible; around 120ms overhead to the pure diagnosis time of 40 minutes.

In our opinion this is an interesting measure showing the limits of using software routines to perform diagnostic procedures and demonstrating the effectiveness of using a HW based strategy for tester implementation.

6.1.3.d Tester memory requirements

The proposed architecture also offers a significant gain in terms of tester memory requirements for pattern storage.

As the diagnosis program is compressed off-line and then decompressed on-the-fly by hardware, only a reduced test set is stored in the tester memory in order to reconstruct the complete pattern, while in traditional approaches commonly the whole test program is transmitted to the tester and stored in its memory in order to be then applied to the DUT.

The test data volume reduction is high, mainly because of horizontal occurrence identified during autonomous memory test execution. Commodities like this are commonly employed in the industrial practice to save tester memory when waiting for test completion; anyway, a minimum additional gain of about 60% is obtained concerning test procedure initialization.

For example, let us consider one initialization and results retrieval step in a backward diagnosis strategy for the memory considered as case study, where the test execution time is neglected.

The original pattern consists of 318 vectors each one including the logical values of 4 signals, finally accounting for a total of 1,272 bits.

With the proposed compression mechanism, the following sequences were identified: six *Vertical*, one *Variable Data Vertical* and one *Output Recording Vertical* occurrences, plus three *Horizontal* and one *Variable Length Horizontal* occurrences. The corresponding TSO file consisted in 29 words, accounting for 336 bits, and the RTS file consisted in 96 bits. So the whole pattern can be stored in 432 bits.

This leads to a 66.03% gain in memory saving, compared to the full original test pattern.

To complete the analysis of the benefit in terms of data volume compression introduced by the illustrated approach, a comparison was done with the solution proposed in [**113**]. In [**113**], a similar compression approach, called reuse, is described where repetitive pattern segments are identified for each signal to be supplied to the DUT, pruned from the original pattern set and saved just once in a mask format. The reuse approach strongly leverage on the pattern regularity, it slices the sequence of stimuli into many parts of the same length and compares them in a vertical manner, eventually reordering them to achieve higher compression rate. This approach was selected for comparison with the proposed methodology since it permits on-the-fly decompression on the tester of the compressed pattern, while this capability is not guaranteed by other compression approaches such as those based on Golomb [**39**] and Huffman [**40**] codes.

To apply the reuse strategy to the considered pattern for embedded memory testing, a preliminary pattern manipulation was requested to slice the pattern in comparable segments of the same length; this additional operation is due to the nature of data that have to be sent to the dBIST for programming a memory test run (i.e., the word for setup the number of test step is 19 bit, while the word to let the dBIST run is 4 bits). Such preliminary manipulation of the pattern slightly enlarged it up to 372 vectors; over this vector set, the reuse strategy achieved a compression of 68.54%, meaning that the final pattern is composed of 468 bits.

By observing this result, it can be stated that the proposed and the reuse strategy achieve comparable results, but the latter imposes additional constraints to the pattern format that may lead to a larger pattern size and consequently to a longer pattern execution. Furthermore, it should be noticed that the reuse approach is not providing the flexibility introduced toward the management of diagnostic flows that is guaranteed by the proposed approach.

6.1.4 Conclusions about embedded memories diagnosis

In this chapter we propose a methodology suitable to perform adaptive diagnosis of SoC embedded memories by means of low-cost test procedures. The described method performs off-line compression and on-the-fly decompression by means of suitable FSMs. It consists in the analysis of the diagnosis program in order to identify special test segments. These segments are used to control the diagnostic flow by hardware. Based on the illustrated schema, the characteristics of an FPGA-based low-cost tester platform are detailed.

Experimental results demonstrate the effectiveness and the feasibility of the described methodology by applying it to embedded memories diagnosis equipped with BIST structures, using a commercial development board including processors, RAM and FPGA resources to implement the tester.

Results show that a high reduction may be obtained in the test/diagnosis time and the tester memory requirements at the cost of a relatively small hardware module to be included in the tester.

6.2 Calibration of MEMS inertial sensors

Being accelerometers and gyroscope MEMS multiple energy domains devices, calibration and testing processes need both electrical and mechanical stimuli. Usually, these two stimuli are generated by independent equipment parts: the so-called rate table, which infuses the movement to the DUT using motors, and the electrical tester (normally including a CPU), which applies the electrical inputs and reads the device outputs, performs the computation needed for calibration and testing, and controls the rate table movements. Usually, the rate

table and the electrical tester are physically separated and connected through wires. These wires are one of the most critical equipment parts for two reasons:

c) since under continuous stress, being twisted and stretched as the rate table rotates in several directions, such wires require much more maintenance than other equipment parts, and this issue is stigmatized when dealing with high testing parallelism rates, where the number of wires could become very large;

d) being long wires, they impose limitations to the electrical stimulation frequency, possibly slowing down the test process and potentially impacting the calibration accuracy.

In the approach proposed in this chapter, we are not intending to add any hardware to the device itself, but to improve the ATS architecture for the packaged MEMS testing step, which can be used in cooperation with the on-chip self-test or the fully electrical approaches. This section presents an innovative MEMS tester architecture suitable for accelerometer and gyroscope MEMS, which moves some of the electrical tester intelligence to the rate table itself. The presented approach is based on a hardware unit to be included in the rate table; this module is able to generate the electrical stimuli and manipulate the device output for autonomously managing the calibration and testing process. The MEMS testing equipment greatly benefits from this architecture since

a) it reduces the amount of data to be transmitted from the distant electrical tester, therefore minimizing wire requirements, especially in case of high testing parallelism

b) trimming calculations are performed directly on the rate table, speeding up this process of orders of magnitude with respect to its software counterpart and overcoming frequency limitations due to wire length.

The resulting tester architecture is therefore superior to existing ones in terms of equipment cost since it guarantees longer life to consumable parts such as wires and, being based on FPGAs, affords scalability and easy reuse of tester resources for several calibration and functional testing issues. Moreover, the proposed tester equipment permits reaching a high degree of parallelism which is actually limited by physical constraints.

6.2.1 Accelerometer and gyroscope MEMS calibration procedure

The MEMS functional testing process is preceded by a calibration, or trimming procedure. Calibration is the process of comparing device outputs with known reference information and determining the coefficients that force the output to match the reference information over a range of output values [**122**]. During this phase, the tester feeds the MEMS component with some trimming values to setup

some internal parameters. Trimming values are not fixed, but have to be determined for each device according to the electrical responses to well defined physical stimulations. First-order bias and scale factor errors are the dominant deterministic elements when speaking about accelerometers and gyroscopes; we consider a linear model for their output.

In (6.4) the linear model is expressed for axis x of an accelerometer:

$$A_x = m_x * G_x + be_x \tag{6.4}$$

where A_x is the output of the device, m_x is the Sensitivity, be_x the Offset error and G_x the acceleration applied to the device on the x axis. For accelerometers, trimming values can be determined using the earth gravity as reference through the four-point tumble method shown in Figure 6.27, as defined in the IEEE Standard Specification Format Guide and Test Procedure for Linear Single-Axis, Nongyroscopic Accelerometers [**123**].

For example, for axis x, the Sensitivity and Offset error can be calculated using (6.5) and (6.6).

$$m_x = \frac{[A_x(90°) - A_x(270°)]}{2} \tag{6.5}$$

$$be_x = \frac{[A_x(0°) + A_x(180°)]}{2} \tag{6.6}$$

where $A_x(90°)$ is the output obtained with sensor axis x in the 90° position (up), $A_x(270°)$ in the 270° position (down), $A_x(0°)$ in 0° position (horizontal) and $A_x(180°)$ is the output obtained with the sensor x axis in the 180° position (horizontal). It is to note that in (6.5) the dividing "2" has [g] units while the dividing "2" in (6.6) is dimensionless. For a triad of orthogonal sensors this should be done for every axis, resulting in a total of six positions.

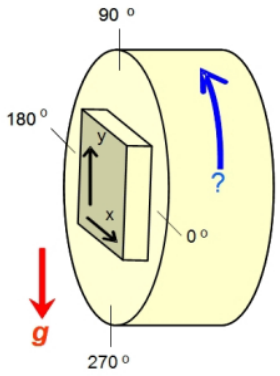

Figure 6.27 Four-point tumble schema for accelerometer calibration.

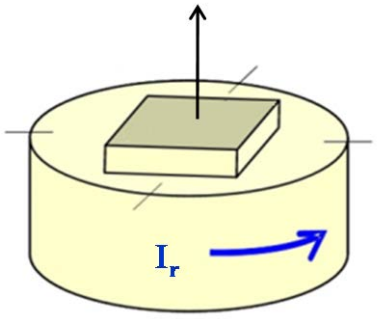

Figure 6.28 Rate table schema for gyroscope calibration.

Gyroscope bias and scale factor trimming values are determined, as defined in the IEEE Standard Specification Format Guide and Test Procedure for Single-Degree-of-Freedom Rate-Integrating Gyros [**124**], using a rate table applying a constant angular rate to the DUT, according to the simplified schema shown in Figure 6.28. A simple way of calculating the bias value (*be*) is by measuring the null output (e_0) when the chip is still. While scale factor (S) is determined by using the rate table as a method for applying an angular rate (I_r), preferably one close to the full scale of the device. With the device spinning at a constant rate, the output is measured (O_r). Bias and scale factor are then determined by (6.7) and (6.8).

$$be = e_0 \tag{6.7}$$

$$S = \frac{(O_r - e_0)}{I_r} \tag{6.8}$$

A single calibration point may be sufficient in some cases. However, more accurate results can be obtained by measuring many points, and performing a linear regression.

Figure 6.29 graphically illustrates the calibration process; once the trimming values are calculated, they are stored inside the DUT.

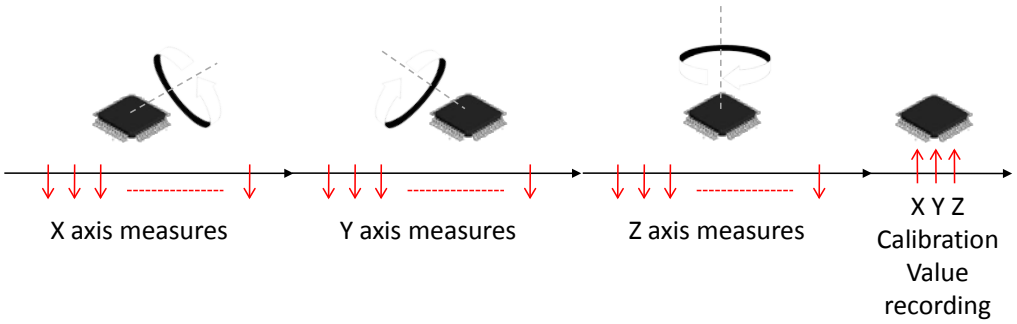

Figure 6.29 Accelerometers and gyroscope calibration flow.

Calibration of both analogue and digital output devices is done following the same general procedures; however, the magnitude of the trimming values changes. For example, an analogue output accelerometer typically has a *Scale Factor* measured in V/g (or mV/g); while in a digital output accelerometer the *Scale Factor* is usually expressed in mg/digit.

The output of analogue MEMS devices mostly corresponds to a voltage, varying in a range according to the chips specifications, while the output of digital devices is communicated using some kind of protocol, often a serial one. This protocol may be a standard one (e.g., SPI, I^2C, or RS-232), or a proprietary one,

defined by the manufacturer. As an example, Figure 6.30 shows a slave four-wire SPI protocol implementation for ATS-DUT communication, which is actually one of the most used communication protocols implemented in MEMS chips.

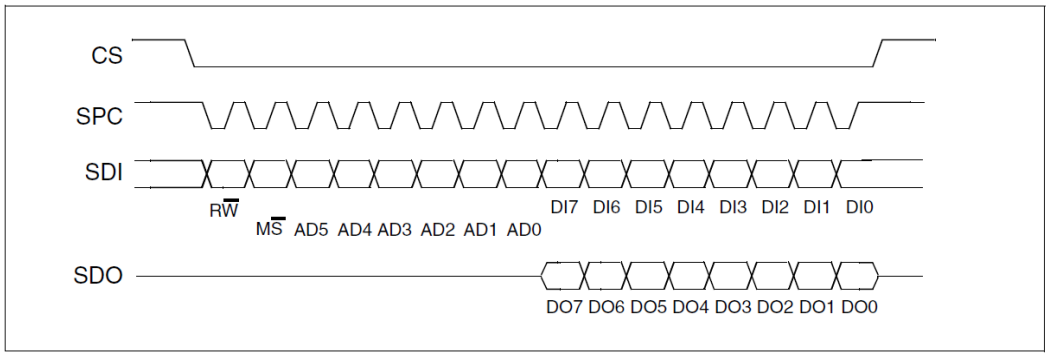

Figure 6.30 Four-wire SPI protocol, 16 bits implementation.

More in detail, the illustrated implementation is used in the ST-LIS331DL accelerometers and ST-L3G4200D gyroscopes, which will be used as case studies later in the paper. Each communication cycle consists of at least two serial bytes. The first bit transmitted on the *Serial Data Input* (SDI) wire indicates if it is a write or read operation, the second is set when more than one read or write commands would address the same register. The next 6 bits encode the register address. Finally, there is one data byte: depending on the communication direction, data go through the *Serial Data Output* (SDO) when data come from the chip or through the SDI signal when data are sent to chip. The *Serial Port Clock* (SPC) signal is the channel's clock, while the *Chip Select* signal (CS) enables the channel and signals to start the communication.

6.2.2 MEMS testing equipment

MEMS testing equipment is composed of two main parts, called rate table and electrical tester. These parts are connected through a number of wires that is currently dependent on the number of pins of the MEMS device and the number of devices accessed in parallel. MEMS equipments may also include some mechanical arms, often called heads, in charge of placing/removing components on/from the rate table; being a complementary equipment commodity, this movement devoted part was not addressed during this thesis work. A MEMS testing equipment conceptual schema is shown in Figure 6.31 detailing the Electrical Tester, the Rate Table and their connection through wires.

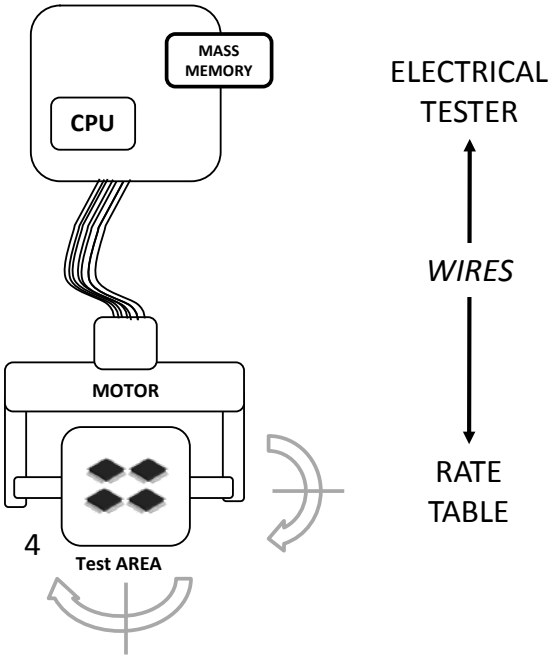

Figure 6.31 MEMS testing equipment conceptual schema.

For industrial calibration, the following process is performed:

1. a number of chips is mounted on the fixture in the test area
2. the rate table rotates as specified by the test receipt
3. concurrently, the electrical tester records the electrical output of the DUTs
4. the registered output data are processed by the CPU existing in the electrical tester in order to calculate the trimming values for each DUT
5. the calculated values are sent back to the DUTs.

After calibration, a test procedure is executed, completing the subsequent steps:

6. another mechanical stimuli is applied by the rate table
7. electrically registration of output is performed by the electrical tester
8. good and faulty devices are marked, and the latter possibly classified according to the test results.
9. the chips are removed from the test area.

For the purpose of this work, an industrial tester with a rate table currently able to test up to 16 MEMS inertial sensors in parallel is considered as a reference. The electrical tester is connected to the rate table with about 100 wires. This great amount of wires is due to the fact that the tester needs to communicate in parallel with each DUT, in order to collect the output data: since the mechanical

movement is unique, all the sixteen DUTs are calibrated and then tested at the same time. After the test is finished, according to the results processed by the tester, the equipment is able to classify the devices in many category bins according to customer requests (e.g., pad, power supply, mechanical failures).

The major purpose of the presented work is to maximize the calibration and test parallelism while mitigating the wire issue and reducing the whole calibration and test procedure length.

6.2.3 Proposed methodology for MEMS calibration and test

The proposed methodology is aimed at enabling high testing parallelism and mitigating mechanical concerns and frequency bottleneck that are affecting current MEMS tester architectures.

In our proposal these objectives are pursued by moving part of the electrical tester functions to the rate table, in particular including stimulation circuitries and trimming calculation units very close to the devices. In this way, the following benefits are achieved:

- Primarily, the number of wires is strongly reduced, and differently from a classical architecture, their number does not grow with increasing parallelism
- Having stimuli application and trimming/functional test calculation performed by circuitries on the rate table
    - the overall process time is significantly reduced
    - the frequency bottlenecks eventually arising because of wire connections are overcome.

Figure 6.32 shows a structural schematic comparison between traditional (a) and proposed (b) MEMS tester architecture. In the proposed architecture, the Rate Table includes an FPGA device, whose content depends on the sequence of electrical stimuli to be applied to the DUTs and on the sequence of values to be observed.

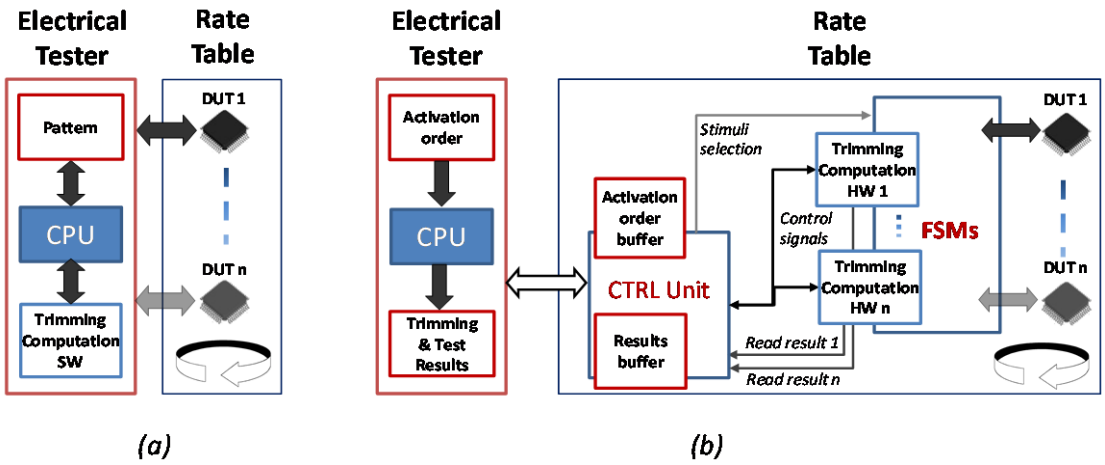

Figure 6.32 Traditional (a) and proposed (b) MEMS tester architecture.

The working principle of the proposed methodology, which is detailed in the next paragraph, is quite simple and it is a viable solution for improving the MEMS tester capabilities.

Firstly, the communication protocol and the trimming and test procedures are analyzed in order to identify recurrent pattern segments; such segments are translated into Finite State Machines (FSMs) to be mapped on the Rate Table FPGA. Thus, calibration and functional test pattern application can be performed through a suitable sequence of commands (or Activation Order), intended to activate different FSMs in a meaningful order, suitable to reproduce the desired sequence of stimuli.

Secondly, a hardware unit is generated and then mapped on the FPGA included in the rate table; this module incorporates the previously identified FSMs. The activation order of the FSMs is stored in an Activation Order buffer initialized by the electrical tester and then read by a Control Unit to execute the calibration and test procedure. The hardware unit is able to manage calibration and functional testing flow adaptively and autonomously by exploiting a trimming computation module. Results are temporarily stored in a Results buffer flushed to the electrical tester at the end of the process.

The suitable circuitries designed to manage the flow are intended to be stored on the FPGA device on the rate table; this feature makes the tester configuration very flexible. In case the communication protocol changes, the only effort to be devised is pattern analysis, which can be also done automatically [**118**]. In case of parallel MEMS testing, the FSMs and the control unit are shared, while the computation unit is replicated for every MEMS, implying that the area occupation grows less than linearly when augmenting the tester parallelism, therefore guaranteeing scalability.

6.2.3.a Stimuli analysis for calibration and functional testing

The proposed approach, intended to work with digital output devices, grounds on the analysis of test pattern regularities [**26**]. The basic principle is to profit from the a priori knowledge of the communication protocol specification and the calibration and testing flow intended to be applied to the DUT. A conceptual diagram of the methodology is shown in Figure 6.33.

An off-line analysis of both the calibration/test flow and the protocol specifications is first done with the purpose of

- identifying recurrent sequences that could then be reproduced by means of proper FSMs
- deriving a suitable activation order for the FSMs, thus correctly reproducing the original calibration/test flow.

The obtained hardware and information resources are then mapped on the Rate Table hardware (specifically on the FPGA device) and stored in memory cores (RAM blocks available on the FPGA). During the calibration and test phase, these resources will allow on-line generation of the proper stimuli for the DUT.

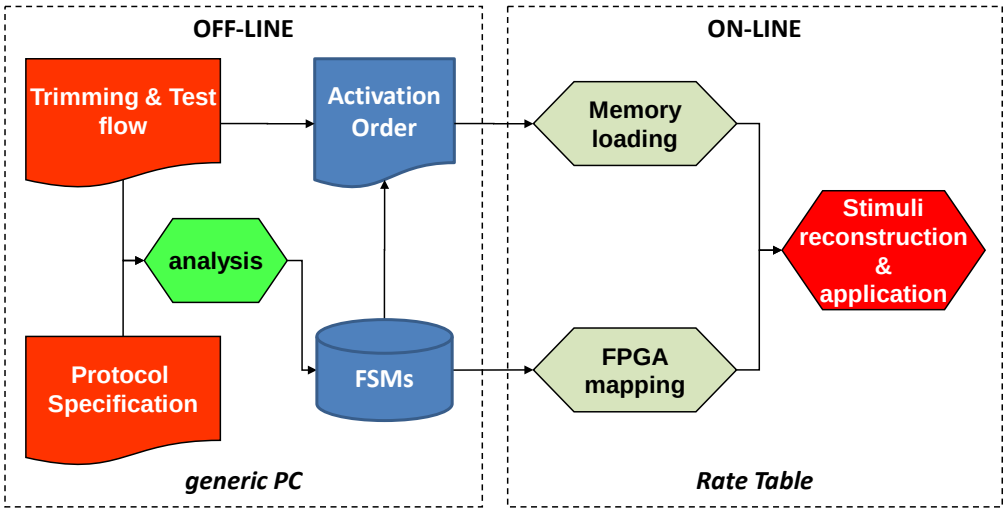

Figure 6.33 Off-line analysis and on-line stimuli reconstruction.

Protocol Aware Sequences

Test data for embedded test execution may be strongly repetitive; for instance, if data/commands are transmitted to an internal memory space to be decoded, the same binary sequence can be sent many times. In general, the repetitiveness of the pattern is also strongly related to the selected access protocol interface [**26**].

The off-line analysis aims at reducing the pattern set size by identifying test segments occurring several times in the considered test set. This phase requires the knowledge of the employed test access mechanism and its inputs are the whole test set description and a set of shorter test segments provided by the test engineer who developed the test recipe. The analysis process [**26**] returns

- the list of test segments in the selected short test segments and identified as hardly recurrent,
- a modified test set description pruned from such recurrent test segments,
- the information needed to rebuild the original test set characteristics in a suitable format.

In the specific case of the MEMS testing, the most frequently adopted access method is the SPI protocol. By analyzing it, five different types of sequences are identified, and then mapped on proper FSMs, in order to be used to reproduce the calibration/test flow. Some kind of occurrences are reproduced without any modifications while others must be completed by input or output values tailored to implement the MEMS testing and calibration flow. The FSMs based approach is scalable and reusable, since it takes advantage of the access protocol repetitiveness and can be easily adapted to different protocols, even the ones showing longer activation sequences.

A *Steady Vertical* (SV) occurrence is encountered when a timing diagram slice is repeated many times in the overall pattern set application, in which all signals to be fed to the DUT have repetitive behaviours. Two examples are shown of Steady Vertical occurrences addressing one specific register in order to perform a write cycle: SV1 (Figure 6.34, a); and a read cycle: SV2 (Figure 6.35, a). These sequences appear several times during the calibration and testing phases and are identified as two different Steady Vertical occurrences.

A *Variable Data Vertical* (VDV) occurrence, on the other hand, is a sequence where all values and timings are repeated, except those related to one signal. VDV1 (Figure 6.34, b) is an example, which may occur when writing the on-line calculated trimming values to the device under test. The variable signal in VDV1 is SDI; the values of the signal, that are marked as **S** in the example of Figure 6.34, are taken from a suitable register (one register per identified VDV sequence) at run-time.

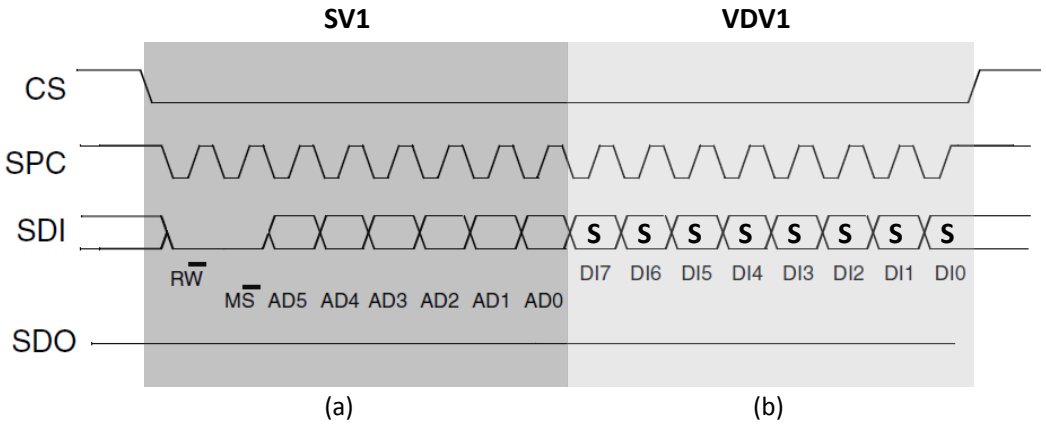

Figure 6.34 Write cycle with two consecutive occurrences: SV (a) and VDV (b).

An *Output Recording Vertical* (ORV) occurrence is a period of time during which the system has to record the output of the DUT. This type of sequence is the one used to read data from the device under test. Data is recorded in a convenient register (one per ORV identified sequence). ORV1 (Figure 6.35, b) is an example where the a priori unknown values that have to be registered from the chip output into a proper register are labelled as **U**.

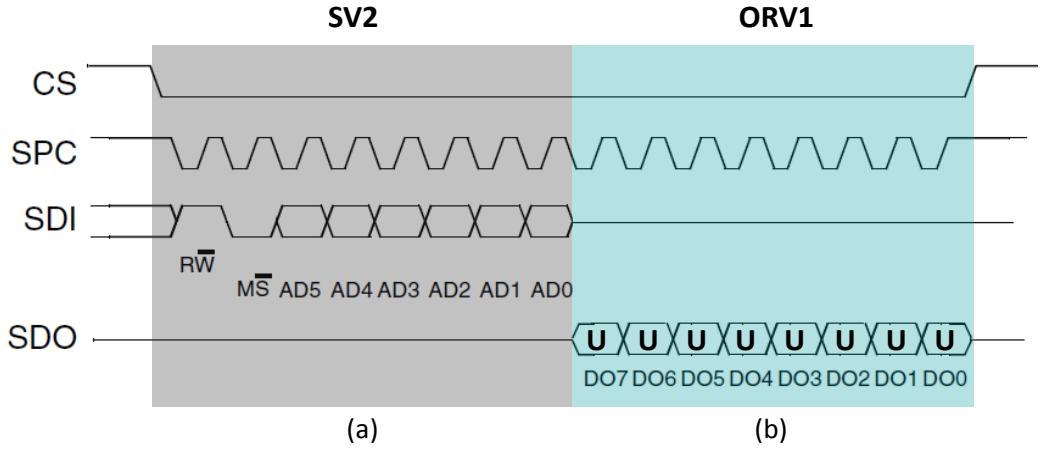

Figure 6.35 Read cycle with two consecutive occurrences: SV (a) and ORV (b).

An *Output Polling Vertical* (OPV) occurrence is defined as a timing diagram slice that at a certain point, marked as **C** in the example shown in Figure 6.36, compares the actual output of the DUT with an expected output, in order to decide whether to repeat the sequence or not. This sequence is usually used for polling the DUT status. MEMS chips may have a setup time prior to giving the correct sensed value and have a dedicated bit (stored in an internal register) that the chip sets when the awaited value is ready. An OPV sequence is used to poll

this bit, so as to know when the output value is valid and should be registered. In the example of Figure 6.36, bit 2 is polled using a read cycle. In the proposed methodology, once an OPV is launched it is repeated until the read value for **C** matches the expected one.

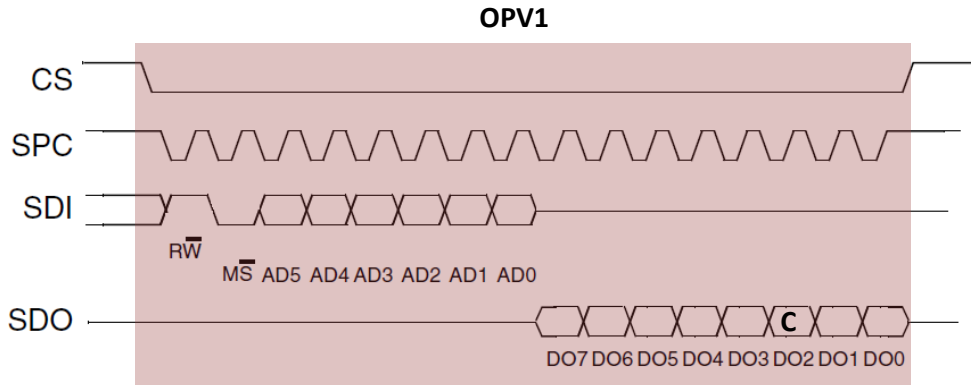

Figure 6.36 Read cycle implementing a polling operation with an OPV occurrence.

Finally, a *Horizontal* (H) occurrence is defined as a period of time during which all signals remain constant. This sequence is usually used for waiting cycles: for example, the system may have to wait for the calculation of the trimming values keeping the signals stable in the meanwhile.

Once all the occurrences in the test set under analysis have been identified, a suitable encoding is needed to further be able to reconstruct the original calibration/test flow. The following encoding has been chosen for these data:

- In case the current pattern segment corresponds to any Vertical occurrence; either Variable Data, Output Recording, Output Polling or a Steady
  - 8 bits are used to describe it in the Activation Order file
  - the MSB of this byte is set to 1
  - the remaining 7 bits indicate the vertical occurrence identification number (up to 128 diverse Vertical occurrences can be identified).
- In case the current pattern segment corresponds to a Horizontal occurrence
  - 16 bits are used to describe its characteristics
  - the MSB of the first byte is set to 0
  - the 2nd to 4th bits provide the horizontal occurrence identification number (up to 7 different horizontal occurrences)
  - the remaining 12 bits store the length in clock cycles of the segment (up to 4096 clock cycles).

Information about special values **S**, **U** and **C** including the signals they are affecting and the expected values when needed are self-contained in the generated FSMs.

Calibration/Test sequence activation order

By exploiting FSMs derived from the identified set of occurrences, and leveraging an ad-hoc trimming hardware (more details are given later about this module), it is possible to reproduce the calibration and functional testing flow. This is done by releasing an activation order of the occurrences to be reproduced based on the described encoding that will finally be loaded on a RAM block available in the Rate Table.

The calibration and test flow typically encompasses five phases, in coordination with the mechanical movements of the Rate Table:

1. Initialization: usually performed once at the beginning just to setup the device by setting some DUT internal configuration registers.
2. Data Collection: the sensor outputs are registered so as to calculate the chip trimming values. The recording process is repeated several times, until sufficient measures are obtained in order to calculate trimming parameters, such as the scale factor and the bias error, with the desired precision. Preliminary computation proactive to trimming value calculation can be even performed concurrently during this phase.
3. Trimming Value Calculation: the DUT is waiting without any stimulation for the computation hardware to complete the trimming value calculation.
4. Calibration: this is the moment when the calculated values are sent to the sensor so that it can save them in its own non-volatile memory.
5. Testing: data are once again collected from the sensor, this time to verify whether they match the expected ones, in order to classify the sensor as good or bad, or even in more subclasses according to the obtained results.

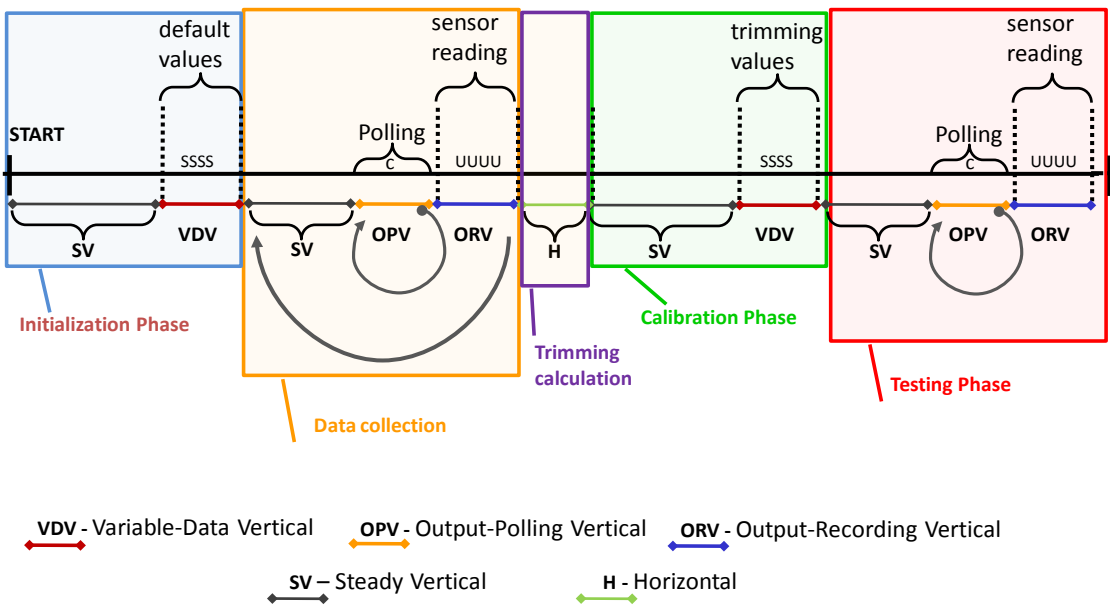

Figure 6.37 Calibration and testing flow implementation by means of FSMs.

An explanatory flow implemented with the defined sequences is described in Figure 6.37. We want to remark that some sequences are reused in different parts of the flow. As an example, the VDV sequence is used at the Initialization phase and during the Calibration phase. In the first case, the variable values **S**, are substituted with some default values; in the latter **S** is substituted with the values calculated in the Calibration phase.

The flexibility of the method for composing the patterns allows describing many different flows with the same FSMs. Moreover, the memory space needed to store the information describing the flow is strongly reduced with respect to the full pattern stored in the traditional tester architecture.

6.2.3.b Parallel MEMS tester architecture

Based on the method described in the previous paragraph, it is possible to design a tester architecture able to decode and reproduce the calibration flow summarized by the produced activation order that acts like a firmware for the circuitries included on the Rate Table.

This architecture, shown in Fig. 13, is composed of two main parts:

- The Stimuli Generator: located in the Rate Table itself, it includes all the modules needed to perform calibration and testing
- The Stimuli Controller: located in the electrical tester. It supervises the entire process controlling both mechanical and electrical stimulation.

In Fig. 13 it is also shown a *Mechanical Synchro block* located in the electrical tester in charge of controlling the motor that moves the Rate Table, coordinated by the Stimuli Controller. In addition, Power driver & Analogue measurements modules are present on the Rate Table, whose purpose is to manage the delivered power and to perform the parametric test on the DUTs; these features are not treated in this paper. Neither are the different stress conditions under which the test shall be performed (e.g., temperature variations); the considered tester takes care of guaranteeing the same scenario for all the MEMS tested in parallel, while keeping the active components of the Rate Table (i.e., Stimuli Generator, Power driver and Analogue measurements modules) at a normal operating environment.

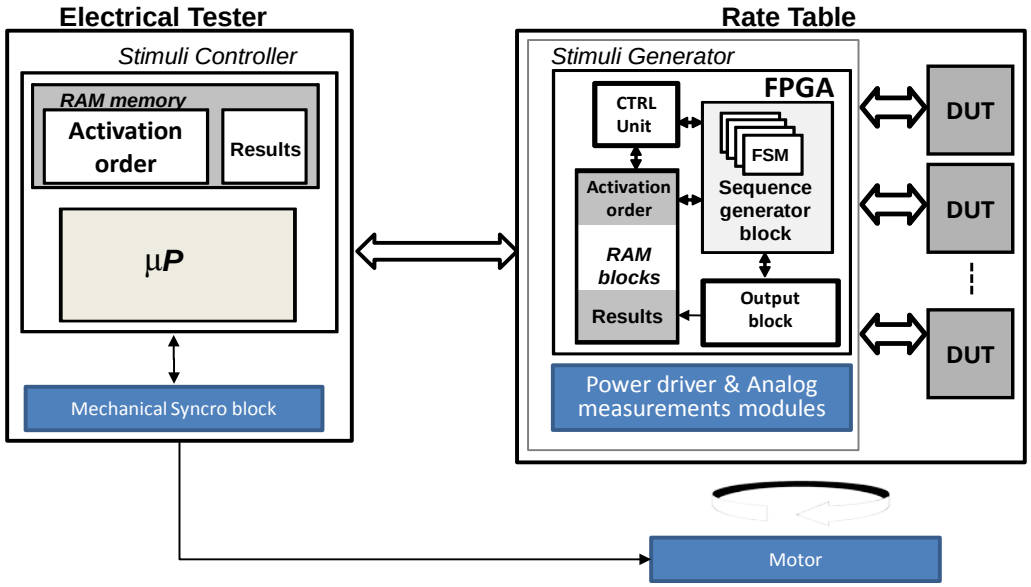

Figure 6.38 Proposed tester architecture, including the two main modules.

The Stimuli Generator performs all the calibration and testing procedures autonomously. It is the most important module of the tester, since it is able to:

- reconstruct the original stimuli based on the activation order stored in the proper RAM block
- complete VDV occurrences, replacing the variable data (S) with values taken from the convenient register (VDV-reg).
- capture the output values of the DUT (U) when ORV occurrences are executed, storing them into the apropos register (ORV-reg) and in the Results RAM block
- perform the polling during the fulfilling of an OPV sequence, comparing the value obtained from the DUT (C) with the expected one, and repeating the OPV in case of mismatch.

- adaptively calculate suitable trimming values.

The Stimuli Controller duties, based on a general purpose microprocessor, are:

- saving the final results sent from the Stimuli Generator when the test is finished
- storing the complete activation order, and sending it to the Stimuli Generator for its loading in the Activation Order RAM block
- coordinating the mechanical movement with electrical stimulation through the Mechanical Synchro block.

For the sake of comprehension, the following paragraphs first describe the Stimuli Generator architecture suitable to calibrate and test a single MEMS chip; afterwards, a parallel architecture is detailed.

Single-chip Rate Table architecture

Figure 6.39 shows in detail the logic of the Stimuli Generator module, located at the Rate Table, which includes four main blocks:

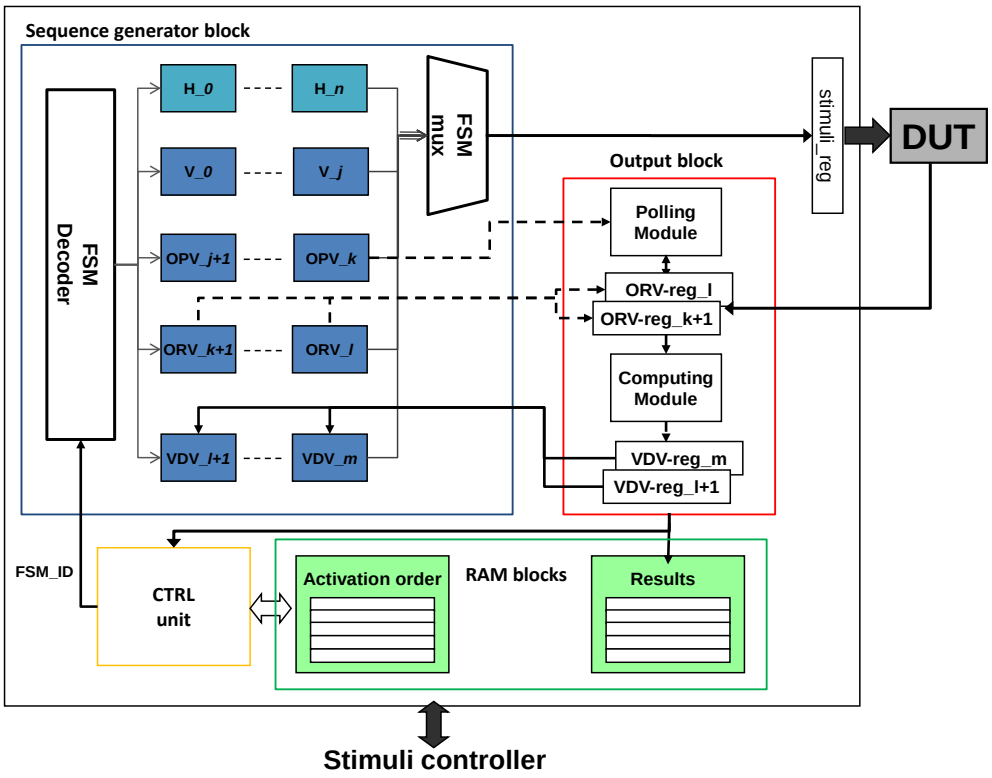

Figure 6.39 Stimuli Generator module schema.

- *Control Unit*: manages the whole process,
  - reading the instructions from the activation order buffer and activating accordingly the appropriate FSM

  - receiving feedback from the Output block, taking appropriate decisions along the calibration flow
  - communicating with the Stimuli Controller to load/unload the RAM blocks and to launch the electrical stimulation.
- *Sequence generator block*: it outputs the values reproduced by the selected FSM, according to the current FSM_ID read in the activation order by the Control Unit, using a decoder and a multiplexer. It works in cooperation with the Output block, by means of one VDV-reg in order to complete each VDV sequence.
- *Output block*: it is devoted to the output manipulation operations; more in particular:
  - checks the test status by observing the C value in the response of OPV occurrences, asking the control unit to repeat the OPV or to continue with the pattern application
  - records the relevant outputs (such as the sensed values) relying on the execution of ORV occurrences and exploiting one ORV-reg per ORV sequence.
  - implements the functions to compute the trimming values and stores the result of the computation in the suitable VDV-reg. The functions shall vary according to the calibration method and the type of DUT.
- *RAM blocks*: these buffers receive the activation order and send the results from and to the *Stimuli Controller*, and are accessed by the *Control Unit* and the *Output block* during the calibration procedure.

Parallel Rate Table architecture

The proposed architecture is intrinsically suitable for dealing with multi-site calibration and testing (i.e. multiple devices are tested in parallel within the same time it would normally take to test one device). If more than one chip is tested in parallel, an *Output block* must be instantiated for each DUT, while a unique *Sequence generator block* is needed. All the chips receive the same stimuli at the same time, but their responses are likely to be different, so the consequent calculated trimming values for each sensor calibration are also likely to be different. This is the reason why the groups of ORV-reg and the VDV-reg are replicated for each device under test. The Computing Module is also replicated to perform the calculations in parallel so as the whole throughput of the system does not fall as it is shown in the experimental results. Anyway, the system will move at the pace of the slowest DUT.

Concerning the Polling Module within the *Output block*, it is also replicated. While it is true that all devices tested in parallel are equivalent in their scope,

working principle and design (i.e., all DUTs are of the same model), they are not identical: during the calibration process each single MEMS device may have a different responding time when accessed via its digital interface. Every chip has to be polled in order to know when the correct output is ready to be read and it is quite common that the number of polling cycles changes from chip to chip. Therefore, some components require more time to be calibrated; the system cannot proceed until all the devices have been calibrated. Hence, in case a polling sequence is activated, the pattern will not progress until data from all DUTs are ready. When this occurs, responses from all chips can be safely registered and the process may continue.

In Figure 6.40, a scheme of the proposed parallel architecture is shown, which is designed to calibrate and test four MEMS devices simultaneously.

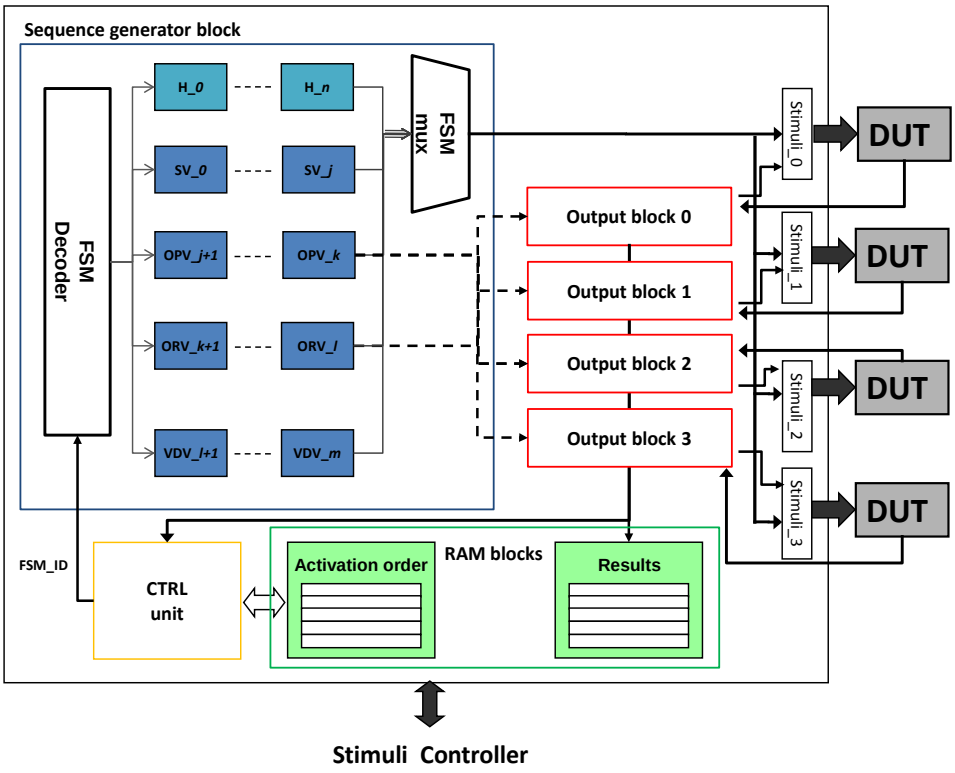

Figure 6.40 Four chips parallel MEMS calibration and testing architecture.

6.2.4 MEMS calibration and testing experimental results

Experimental results have been collected for two commercial chips, the LIS331DL 3-axis accelerometer and the L3G4200d 3-axis gyroscope by STMicroelectronics.

The **LIS331DL accelerometer** [**125**] is a 3-axis accelerometer featuring digital SPI/I2C serial interface standard output. This component is capable of detecting

acceleration in the range ±8g with a maximum data rate of 4MHz; it uses a Land Grid Array (LGA) package. Calibration and functional test is performed by moving the chip in six positions according to the standard procedure described in section 6.2.1, which means that proper measurements are done when final positions are reached.

The **L3G4200d gyroscope** [**126**] is a 3-axis gyroscope including an interface able to provide the detected angular rate to the external world through a digital SPI/I2C interface. This gyroscope is capable of performing measures up to 2,000 degrees per second (dps) with a maximum data rate of 10MHz and uses an LGA package. Calibration and functional test is performed by applying an angular rotation corresponding to near 2,000 dps along the three axes, one at a time. Measurements are performed at a constant rotational rate, i.e., after the component is accelerated by the rate table.

6.2.4.a Calibration and functional testing flow analysis

For both devices, a calibration/testing flow analysis was performed resulting in the definition of a set of occurrences including fifteen Steady Vertical, three Output Recording Vertical, three Output Polling Vertical and six Variable Data Vertical sequences. These identified sequences have been translated into FSMs as previously described; most of these sequences are shared among the accelerometer and the gyroscope, while some Steady Vertical sequences are unique for each chip.

The activation order to perform a single measure for one axis for the accelerometer is finally composed of 17 8-bits words; completing a calibration/functional testing flow corresponding to 64 measurements per axis requires 203 words, corresponding to the execution of a 2,144 clock cycles long procedure. For the gyroscope the number of Activation Order words describing the flow is lower due to the different method of calibration; to perform a single measure 12 words are required, while the complete flow requires 105 sequences, performing 32 measures per axis.

As it is further detailed in the next paragraph, some memory is required on the rate table to store the information concerning the activation order, which is about 600 and 300 bytes large for the accelerometer and the gyroscope, respectively.

6.2.4.b Rate Table characteristics

As described in section 6.2.3.b , the architecture of the Rate Table in the proposed version includes an FPGA chip which is in charge of storing the calibration/testing hardware, i.e., the FSMs and the trimming computing module.

In the current implementation, the used FPGA is a device of the XILINX VIRTEX family.

Resources occupied on the FPGA to perform calibration and testing of a single MEMS device account for 890 4-input LUTs, 978 flip-flops and 2 BRAM blocks of 18 Kb, reaching a working frequency of 150MHz. When dealing with parallel Rate Table architecture, achieving an 8 DUT parallelism means increasing the FPGA occupation to 2,268 LUTs and 5,346 flip-flops, while the number of RAM blocks is left unchanged. TABLE XVIII details the occupation of the different Rate Table modules stored in the FPGA.

TABLE XVIII FPGA REQUIREMENTS OF THE DEVELOPED ARCHITECTURE

	Rate Table architecture			
	Single MEMS		**8 MEMS**	
Module	**LUTs**	**FFs**	**LUTs**	**FFs**
Sequence generator block	594	310	594	310
Output block	218	624	1,580	4,992
Control Unit	78	44	94	44
RAM blocks	2 * 18Kb BRAM			

The results show the scalability of the proposed architecture; in fact, moving from a single MEMS device to a parallel architecture implies an increase in the *Output block*, while the *Sequence generator block*, the *Control Unit* and the *RAM blocks* maintain the same size. When changing the MEMS under test, it is possible to reuse the *Sequence generator block* or the *Output block* if the communication protocol or the calibration/testing algorithm do not change, respectively.

A prototypical implementation of the Rate Table with a 128 MEMS parallelism has been also evaluated, where the discussed eight devices module on FPGA is replicated sixteen times in sixteen different PCB boards, each including an FPGA. The size of this Rate Table is about 30x30x15 cm.

6.2.4.c Measured benefits

The benefit introduced by the illustrated architecture is manifold.

Wire reduction. With respect to the traditional architecture generating stimulation using the electrical tester only, there is a significant wire reduction. This advantage derives from the minimization of the data to be transmitted between the rate table and the electrical tester, since the rate table directly generates the input patterns and records/processes the output. While at least 6 wires are needed in the traditional architecture to stimulate each chip, and this number increases linearly with the number of DUTs to be tested in parallel, a few

wires are needed in the proposed architecture just to transmit the sequence order for activating the FSMs prior to the procedures execution. Similarly, the outputs generated during the procedure execution are sent to the electrical tester only at the end of the procedure. In our design, the control unit implements a proprietary communication protocol based on 5 data wires; adding power supply and mechanical synchronization related wires, the wire count finally reaches 11.

Frequency requirements mitigation. The proposed approach also eliminates the frequency constraints related to wires, since there is no need for high transmission rates, such as those required in the traditional architecture when testing chips with fast communication requirements. In fact, communication is implemented only before the calibration and testing procedure execution to load the activation order, and at the end of the procedure to flush the result buffer. For the sake of synchronization no stringent frequency requirements have to be respected.

Test Data Volume reduction. Thanks to the communication protocol analysis producing FSM, a small memory space is required on the Rate Table since the Activation Order is composed of a few words. With respect to a complete pattern fully stored as a software sequence of bits, the reduction ratio may reach up to about 81%, as disclosed in TABLE XIX.

TABLE XIX COMPARISON OF THE PATTERN MEMORY OCCUPATION BETWEEN PROPOSED (*new*) AND TRADITIONAL (*old*) TESTING/CALIBRATION METHODS

	accelerometer			**gyroscope**		
	pattern size		**Volume**	**pattern size**		**Volume**
	old	*new*	**reduction**	*old*	*new*	**reduction**
single measure	80	17	78 %	64	12	81 %
whole	1072	203	81 %	560	105	81 %

Test Time reduction. In the traditional architecture, measurements are performed by the electrical tester and trimming computation executed by software routines; this approach implies long calculations, since software is usually slow and in case of parallel testing the time grows linearly since the computation has to be repeated sequentially for each DUT. In the proposed approach, the trimming time bottleneck is overcome since the trimming computation is performed directly on the Rate Table by hardware circuitries taking a few high-speed clock cycles. Furthermore, having the trimming computation hardware replicated for each DUT, the process is performed simultaneously on a set of devices. TABLE XX shows some figures for the Trimming Computation time requested by the traditional and the proposed architecture; different parallelism rates are considered for the both approaches. For each architecture, the time for one

calculation is measured, the performed calculation is the mean value of two read samples; the proposed architecture is running at a 150MHz frequency while the traditional architecture includes a 2.66GHz processor.

TABLE XX NUMBER OF WIRES AND TRIMMING COMPUTATION TIME COMPARISON BETWEEN TRADITIONAL (OLD) AND PROPOSED (NEW) WITH DIFFERENT PARALLELISM RATES.

		1 DUT		**8 DUT**		**128 DUT**	
		old	*new*	*old*	*new*	*old*	*new*
wires		6	11	48	11	768	11
trimming computing time	*64 measures/chip*	6.4 ms	427 ns	51.2 ms	427 ns	819 ms	427 ns
	2 measures/chip	0.2 ms	14 ns	1.6 ms	14 ns	25.6 ms	14 ns

We can observe that the number of wires was reduced by more than 75% when testing MEMS in parallel and that calibration calculations speed was increased by at least four orders of magnitude in all cases.

6.2.5 Conclusions about MEMS calibration and testing

This chapter provides a methodology suitable to increase tester parallelism and speed up the trimming and testing procedure for digital output MEMS inertial sensors. The proposed method consists in the analysis of the test pattern set for identifying some special test segments. These segments are pruned from the pattern set and reproduced in a hardware mapped on the tester Rate Table; therefore, the described strategy relies on the assumption that a FPGA is included in the used tester architecture to store the suitable stimuli generator.

Experimental results show the effectiveness and the feasibility of the methodology for the speed-up and massive parallelism of test and calibration process for inertial MEMS sensors needing both mechanical and electrical stimuli, with a low HW overhead.

Chapter 7
Conclusions

Semiconductor devices are present in our daily life, from cars to cell phones, from oil industry to entertainment. While devices get smaller and cheaper, we expect them to outperform their predecessors in terms of speed, memory capacity and power consumption. However, we all give for granted that they will perform their function correctly, according to their design. This property is called reliability. In order to assess its reliability, the device is subject to a series of tests in different moment throughout its lifetime. Manufacturing testing and online testing is a possible coarse classification for semiconductor devices testing. The former performed before entering the market; the latter when the device is already in use in its final application.

The work done within this thesis aimed at providing new techniques for reliability characterization of electronic circuits. In particular, we focused on low cost testing approaches, which profit from standard protocols and test access mechanisms. The mains target of the work are SoC embedded devices, as memories, microprocessor and mixed signal devices. Still, the proposed techniques are also applicable for stand-alone devices.

In the first part of this work we concentrated in using the microprocessor resources to test itself. We developed new software-based self-test programs targeting non-easily-testable pipelined microprocessor modules, as prediction units, address adders and data forwarding pipeline interlock mechanisms. All three modules have strong consequences in the microprocessor behaviour, whether because they lead to producing wrong outputs or to performance degradation. On the other hand, they are modules not directly accessed from outside the microprocessor. Carefully crafted routines were developed and experimented in both academic and industrial case studies with successful results.

Secondly, a new concept for on chip testing was introduced. An infrastructure Intellectual Property, which is able to generate test patterns and evaluate device responses on-the-fly. The proposed concept keep most of the software-based self-test advantages, like allowing at-speed testing and not requiring an

expensive external tester. On the other hand it helps in overriding some of its probable drawbacks, such as big memory footprint, long testing times and intrusion into the system memory resources. Results in microprocessor testing are favourable, reducing test time (85%) and program length (90%); whereas its cost is affordable (1,3% of the overall system area). The infrastructure was also proved to be useful in implementing embedded memory tests.

Finally, we focused on procedures that are not (at least at the current state of the art) possible to perform without the aid of an external Automatic Test Equipment (ATE), like diagnosis or calibration procedures. We developed an embedded memories diagnosis methodology, which enables the implementation of backward, forward and pause & resume diagnosis algorithms. The methodology performs off-line compression and on-the-fly decompression and leads to more 60% memory savings in the performed experiments. A similar approach was adapted to create a tester module that allows parallel testing of MEMS inertial sensors. This application is particularly sensible to parallelism, since the devices require mechanical stimuli to be calibrated and tested, but this stimuli stresses the cables connecting moving and fixed parts of the ATE. The approach allowed the implementation of a 128 parallelism rate with 75% less wires than if traditional approach was used (actually, the traditional method is not practically possible, because the number of wires would be prohibitive).

Concluding, we proposed, implemented and evaluated novel algorithms for reliability characterization of electronic circuits, presenting results never appeared in the literature before.

Bibliography

[1] ITRS, "International Technology Roadmap for Semiconductors," The International Technology Roadmap for Semiconductors, 2012.

[2] IEEE, "610-1991 - IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries," IEEE Computer Society Standard ISBN 1-55937-079-3, 1991.

[3] E. Halley, "An estimate of the degrees of the mortality of mankind, drawn from curious tables of the births and funerals at the city of Breslau; with an attempt to ascertain the price of annuities upon lives," *Philosophical Transactions of the Royal Society*, vol. 17, pp. 596-610, Jan. 1693.

[4] IEC, "IEC 60300-1 - Dependability management," International Electrotechnical Commission Standard, 2003.

[5] IEC, "60050 191-02-03 - International Electrotechnical Vocabulary," International Electrotechnical Commission Standard, 1990.

[6] IEC, "IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems," International Electrotechnical Commission Standard, 2010.

[7] ISO, "26262 - Road vehicles - Functional safety standard," International Organization for Standarization Standard, 2011.

[8] CENELEC, "prEN 50126-1:2012 - Railway applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS)," European Committee for Electrotechnical Standarization Standard (to be voted), 2012.

[9] CENELEC, "prEN 50126-4:2012 (21754) - Railway applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) - Part 4: Functional Safety - Electrical/Electronic/Programmable electronic systems," European Committee for Electrotechnical Standarization Standard (to be voted), 2012.

[10] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114-117, Apr. 1975.

[11] E. J. McCluskey and J. F. Poage, "Derivation of optimum test sequences for

sequential machines," in *Proceedings of the Fifth annual Symposium on Switching Theory and Logical Design*, 1964, pp. 121-132.

[12] E. B. Eichelberger and T. W. Williams, "A logic Design Structure for LSI testability," *Journal of Design Automation & Fault-tolerant Computing*, vol. 2, pp. 165-178, May 1978.

[13] IEEE, "1149.1-2001 - IEEE Standard Test Access Port and Boundary-Scan Architecture," IEEE Computer Society ISBN 0-7381-2944-5, 2001.

[14] J. P. Hayes and A. D. Friedman, "Test point placement to simplify fault detection," *IEEE Tansactions on Computers*, vol. 23, no. 7, pp. 727-735, Jul. 1974.

[15] S. N. Bhatt, F. R. K. Chung, and A. L. Rosenberg, "Partitioning circuits for improved testability," *Algorithmica*, vol. 6, no. 1-6, pp. 37-48, Jun. 1991.

[16] E. J. McCluskey, "Built-in self-test techniques," *IEEE Design & Test of Computers*, vol. 2, no. 2, pp. 21-28, Apr. 1985.

[17] V. D. Agrawal, C. R. Kime, and K. K. Saluja, "A tutorial on built-in self-test. Part 1: Principles," *IEEE Design & Test of computers*, vol. 10, no. 1, pp. 73-82, Mar. 1993.

[18] W. H. McAnney and J. Savir, "Built-in checking of the correct self-test signature," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1142-1145, Sep. 1988.

[19] M. Abramovici, C. E. Stroud, and J. M. Emmert, "Online BIST and BIST-based diagnosis of FPGA logic blocks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 12, pp. 1284-1294, Dec. 2004.

[20] A. Manzone, et al., "Integrating BIST techniques for on-line SoC testing," in *Proceeding of the IEEE International On-Line Testing Symposium*, 2005, pp. 235-240.

[21] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor Software-Based Self-Testing," *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4-19, Jun. 2010.

[22] P. K. Parvathala, K. Maneparambil, and W. C. Lindsay, "Functional random instruction testing (FRITS) method for complex devices such as microprocessors," U.S. Patent 6948096, Sep. 2005.

[23] I. Bayraktaroglu, J. Hunt, and D. Watkins, "Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues," in *Proceedings of the International Test Conference*, Santa Clara, 2006, pp. 1-7.

[24] J. Crafts, et al., "Testing the IBM Power 7 TM 4 GHz Eight Core Microprocessor," in *Proceedings of the International Test Conference*, Austin, 2010, pp. 1-10.

[25] H. Al-Asaad, B. T. Murray, and J. P. Hayes, "Online BIST for embedded systems," *Design & Test of Computers*, vol. 15, no. 4, pp. 17-24, Oct. 1988.

[26] P. Bernardi and M. Sonza Reorda, "A novel methodology for reducing SoC test data volume on FPGA-based testers," in *Proceedings on Design, Automation and Test in Europe*, Munich, 2008, pp. 194-199.

[27] D. Appello, P. Bernardi, R. Cagliese, M. Giancarlini, and M. Grosso, "An innovative and low-cost industrial flow for reliability characterization of SoCs," in *Proceedings of the European Test Symposium*, Verbania, 2008, pp. 140-145.

[28] R. Kapur, R. Chandramouli, and T. W. Williams, "Strategies for low-cost test," *IEEE Design & Test of Computers*, vol. 18, no. 6, pp. 47-54, Dec. 2001.

[29] H. Yi, J. Song, and S. Park, "Low-cost scan test for IEEE-1500-based SoC," *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 5, pp. 1071-1078, May 2008.

[30] K. Chakrabarty, "Low-cost modular testing and test resource partitioning for SoCs," *IEE Proceedings Computer & Digital Techniques*, vol. 152, no. 3, pp. 427-441, May 2005.

[31] P. T. Gonciari, B. M. Al-Hashimi, and N. Nicolici, "Integrated test data decompression and core wrapper design for low-cost system-on-a-chip testing," in *Proceedings of the International Test Conference*, 2002, pp. 64-73.

[32] M. Beck, O. Barondeau, F. Poehl, X. Lin, and R. Press, "Measures to improve delay fault testing on low-cost testers—A case study," in *Proceedings of the VLSI Test Symposium*, 2005, pp. 223-228.

[33] IEEE, "1500-2005 - IEEE Standard Testability Method for Embedded Core-based Integrated Circuits," Test Technology Technical Council of the IEEE Computer Society Standard E-ISBN 978-0-7381-4694-2, 2012.

[34] E. J. Marinissen and M. Lousberg, "The role of test protocols in testing embedded-core-based system ICs," in *Proceedings of the European Test Workshop*, Constance, 1999, pp. 70-75.

[35] Y. Zorian, "Guest editor's introduction: What is infrastructure IP?," *IEEE Design & Test of Computers*, vol. 19, no. 3, pp. 3-5, Jun. 2002.

[36] D. Gizopoulos, et al., "Systematic Software-Based Self-Test for Pipelined Processors," *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, vol. 16, no. 11, pp. 1441-1453, Nov. 2008.

[37] A. Benso, S. Chiusano, S. Di Carlo, P. Prinetto, and F. Ricciato, "HD2BIST: A hierarchical framework for BIST scheduling, data patterns delivering and diagnosis in SoCs," in *Proceedings of the International Test Conference*, Atlantic City, 2000, pp. 892-901.

[38] C. E. Stroud, *A Designer's Guide to Built-in Self-Test*. United States of America: Kluwer Academic Publishers - Springer, 2002.

[39] A. Chandra and K. Chakrabarty, "System-on-a-chip test-data compression

and decompression architectures based on Golomb codes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 355-368, Mar. 2001.

[40] P. T. Gonciari, B. M. Al-Hashimi, and N. Nicolici, "Variable-length input Huffman coding for system-on-a-chip test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 783-796, Jun. 2003.

[41] N. Mukherjee, A. Pogiel, J. Raijski, and J. Tyszer, "High volume diagnosis in memory BIST based on compressed failure data," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 441-453, Mar. 2010.

[42] S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors," *IEEE Transactions on Computers*, vol. 29, no. 6, pp. 429-441, Jun. 1980.

[43] R. G. Daniels and W. C. Bruce, "Built-In Self-Test Trends in Motorola Microprocessors," *IEEE Design & Test of Computers*, vol. 2, no. 2, pp. 64-71, Apr. 1985.

[44] G. Giles, "Is scan (alone) sufficient to test today's microprocessors? Not quite, but we can't get the job done without it," in *Proceedings of the International Test Conference*, 2002, p. 1197.

[45] Y. S. Chang, S. Chakravarty, H. Hoang, N. Thorpe, and K. Wee, "Transition tests for high performance microprocessors," in *Proceedings of the VLSI Test Symposium*, 2005, pp. 29-34.

[46] L. Chen and S. Dey, "DEFUSE: a deterministic functional self-test methodology for processors," in *Proceedings of the VLSI Test Symposium*, Montreal, 2000, pp. 5-262.

[47] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-Based Self-Testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461-475, Apr. 2005.

[48] M. Mayberry, J. Johnson, N. Shahriari, and M. Tripp, "Realizing the benefits of structural test for Intel microprocessors," in *Proceedings of the International Test Conference*, 2002, pp. 456-463.

[49] M. Tripp, S. Picano, and B. Schnarch, "Drive Only at Speed Functional Testing; one of the techniques Intel is using to control test costs," in *Proceedings of the International Test Conference*, Austin, 2005, pp. 136-143.

[50] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of Software-Based Self-Test procedures for dependable automotive applications," in *IEEE Design, Automation and Test in Europe*, Grenoble, 2011, pp. 1-2.

[51] K. Batcher and C. Papachristou, "Instruction randomization self test for

processor cores," in *Proceedings of the VLSI Test Symposium*, Dana Point, 1999, pp. 34-40.

[52] P. Bernardi, et al., "On-Line Software-Based Self-Test of the Address Calculation Unit in RISC Processors," in *Proceedings of the European Test Symposium*, Annecy, 2012, pp. 1-6.

[53] G. Hetherington, et al., "Logic BIST for large industrial designs: real issues and case studies," in *Proceedings of the International Test Conference*, Atlantic City, 1999, pp. 358-367.

[54] T. Powell, A. Kumar, J. Rayhawk, and N. Mukherjee, "Chasing Subtle Embedded RAM Defects for Nanometer Technologies," in *Proceedings of the International Test Conference*, Austin, 2005, pp. 850-858.

[55] Z. Conroy, G. Richmond, X. Gu, and B. Eklow, "A Practical Perspective on Reducing ASIC NTFs," in *Proceedings of the International Test Conference*, Austin, 2005, pp. 349-355.

[56] J. B. Khare, A. B. Shah, A. Raman, and G. Rayas, "Embedded Memory Field Returns - Trials and Tribulations," in *Proceedings of the International Test Conference*, Santa Clara, 2006, pp. 1-6.

[57] A. van de Goor, S. Hamdioui, and G. Gaydadjiev, "Using a CISC microcontroller to test embedded memories," in *Proceedings of the IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, Vienna, 2010, pp. 261-266.

[58] A. van de Goor, G. Gaydadjiev, and S. Hamdioui, "Memory testing with a RISC microcontroller," in *Proceedings on Design, Automation and Test in Europe*, Dresden, 2010, pp. 214-219.

[59] A. J. van de Goor, S. Hamdioui, and H. Kukner, "Generic, Orthogonal and Low-cost March Element based Memory BIST," in *Proceedings of the International Test Conference*, Anaheim, 2011, pp. 1-10.

[60] X. Du, N. Mukherjee, W. T. Cheng, and S. Reddy, "Full-Speed Field-Programmable Memory BIST Architecture," in *Proceedings of the International Test Conference*, Austin, 2005, pp. 1173-1181.

[61] Y. Park, J. Park, T. Han, and S. Kang, "An Effective Programmable Memory BIST for Embedded Memory," *IEICE Transactions on Information and Systems*, vol. 92, no. 12, pp. 2508-2511, Dec. 2009.

[62] M. Burns and G. Roberts, *An introduction to mixed-signal IC test measurement*. Oxford University Press, 2001.

[63] IEEE, "1241-2000 - IEEE Standard for Terminology and Test Methods for Analog-To-Digital Converters," IEEE Standard E-ISBN 0-7381-2725-6, 2001.

[64] L. Rolindez, S. Mir, J. L. Carbonero, D. Goguet, and N. Chouba, "A sterea ΣΔ ADC architecture with embedded SNDR self-test," in *Proceedings of the*

IEEE International Test Conference, Santa Clara, 2007, pp. 1-10.

[65] IEEE, "1057-2007 - IEEE Standard for Digitizing Waveform Recorders," IEEE Standard E-ISBN 978-0-7381-5350-6, 2008.

[66] H. Mattes, S. Sattler, and C. Dworski, "Controlled sine wave fitting for ADC test," in *Proceedings of the International Test Conference*, 2004, pp. 963-971.

[67] H. C. Hong, F. Y. Su, and S. F. Hung, "A fully integrated built-in self-test ΣΔ ADC based on the modified controlled sine-wave fitting procedure," *Transactions on Instrumentations and Measurement*, vol. 59, no. 9, pp. 2334-2344, Sep. 2010.

[68] N. Chouba and L. Bouzaida, "A BIST architecture for sigma delta ADC testing based on embedded NOEB self-test and CORDIC algorithm," in *Proceedings of the International Conference on Design and Technology of Integrated Systems in Nanoscale Era*, 2010, pp. 1-7.

[69] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on Electronic Computing*, vol. EC-8, no. 3, pp. 330-334, Sep. 1959.

[70] L. Robin, "MEMS Accelerometer, Gyroscope and IMU market 2008-2013," Yole Déeveloppment, 2009.

[71] M. F. Cortese and G. Avenia. (2010, Jul.) Electronic Products. [Online]. http://www.electronicproducts.com/Test and Measurement/Benchtop Rack Mountable/MEMS testing innovations in mass production.aspx?terms=MEMS%20testing%20innovations%20in%20mass%20production

[72] H. Farahani, J. K. Mills, and W. L. Cleghorn, "Design, fabrication and analysis of micromachined high sensitivity and 0% cross-axis sensitivity capacitive accelerometers," *Microsystem Technologies*, vol. 15, no. 12, pp. 1815-1826, Sep. 2009.

[73] H. Liu, S. Gao, X. Liang, and L. Jin, "Performance analysis and measurement of micro-machined gyroscope," in *Proceedings of the International Conference on Electronic Measurement & Instruments*, Beijing, 2009, pp. 30-34.

[74] A. M. Shkel, C. Acar, and C. Painter, "Two types of micromachined," in *Proceedings of IEEE Sensors*, Irvine, 2005, pp. 531-536.

[75] H. Cheng, Y. Zhao, B. Qiang, and Y. Liu, "Design of testing system for accelerometer based on GP-IB," in *Proceedings of the International Conference on Electronic Measurement & Instruments*, Beijing, 2009, pp. 548-551.

[76] V. Skvortzov, Y. C. Cho, B.-L. Lee, and C. Song, "Development of a Gyro Test System at Samsung Advanced Institute of Technology," in *Proceedings of the Position Location and Navigation Symposium*, 2004, pp. 133-142.

[77] Z. Liang and J. Zhang, "A dynamic balance testing framework for gyroscope based on embedded system," in *Proceedings of the International Conference on Artificial Intelligence and Computational Intelligence*, Shanghai, 2009, pp. 573-577.

[78] H. V. Allen, S. C. Terry, and D. W. de Bruin, "Self-testable accelerometer systems," in *Proceedings of the IEEE Micro Electro Mechanical Systems*, Salt Lake City, 1989, pp. 113-115.

[79] B. Charlot, S. Mir, F. Parrain, and B. Courtois, "Electrically induced stimuli for MEMS self-test," in *Proceedings on VLSI Test Symposium*, Marina del Rey, 2001, pp. 210-215.

[80] N. Deb and R. D. Blanton, "Built-in self test of CMOS-MEMS accelerometers," in *Proceedings of the International Test Conference*, 2002, pp. 1075-1084.

[81] A. Dhayni, S. Mir, L. Rufer, and A. Bounceur, "Pseudorandom functional BIST for linear and nonlinear MEMS," in *Proceedings on Design, Automation and Test in Europe*, Munich, 2006, pp. 1-6.

[82] X. Xiong, Y.-L. D. Wu, and W.-B. Jone, "A dual-mode built-in self-test technique for capacitive MEMS Devices," *IEEE Transactions on Instrumentation and Measurement*, vol. 54, no. 5, pp. 1739-1750, Oct. 2005.

[83] R. Ramadoss, R. Dean, and X. Xiong, "MEMS Testing," in *System-on-Chip Test Architectures: Nanometer Design for Testability*. Brulington, United States: Morgan Kaufmann, 2008, ch. 13, pp. 591-651.

[84] N. Dumas, F. Azais, F. Mailly, and P. Nouet, "Study of an electrical setup for capacitive MEMS accelerometers test and calibration," *Journal of Electronic Testing, theory and applications*, vol. 26, no. 1, pp. 111-125, Feb. 2010.

[85] S. McFarling, "Combining branch predictors," Digital Equipment Corporation Western Research Laboratories Technical Note, 1993.

[86] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, vol. 17, no. 1, pp. 6-22, Jan. 1984.

[87] M. Hatzimihail, M. Psarakis, D. Gizopoulos, and A. Paschalis, "A methodology for detecting performance faults in microprocessors via performance monitoring hardware," in *Proceedings of the International Test Conference*, Santa Clara, 2007, pp. 1-10.

[88] E. Sanchez, M. Sonza Reorda, and A. Tonda, "On the functional test of Branch Prediction Units based on Branch History Table," in *Proceedings of IEEE/IFIP 19th International Conference on VLSI and System-on-Chip*, Hong Kong, 2011, pp. 278-283.

[89] Freescale Semiconductor, *e200z6 PowerPC (TM) Core Reference Manual*.

2004.

[90] H. Grigoryan, G. Harutyunyan, S. Shoukourian, V. Vardanian, and Y. Zorian, "Generic BIST architecture for testing of content addressable memories," in *Proceedings of the IEEE 17th International On-Line Testing Symposium*, Athens, 2011, pp. 86-91.

[91] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64-71, Jul. 2002.

[92] J. W. Perez, J. Velasco-Medina, D. Ravotto, E. Sanchez, and M. Sonza Reorda, "A hybrid approach to the test of cache memory controllers embedded in SoCs," in *Proceedings of the IEEE International On-line testing symposium*, Rhodes, 2008, pp. 143-148.

[93] Opencores. (2009) opencores.org. [Online]. http://www.opencores.org,minimips

[94] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transaction on Computer Aided Design of Integrated Circuits*, vol. 24, no. 1, pp. 88-99, Jan. 2005.

[95] J. Shen and A. Abraham, "Synthesis of Native Mode Self-Test Programs," *Journal of Electronic Testing: Theory and Applications*, vol. 13, no. 2, pp. 137-148, Oct. 1998.

[96] E. Sanchez, M. Sonza Reorda, and G. Squillero, "On the transformation of manufacturing test sets into on-line test sets for microprocessors," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2005, pp. 494-502.

[97] A. Merenitis, G. Theodorou, M. Giorgaras, and N. Kranitis, "Directed Random SBST Generation for On-Line Testing of Pipelined Processors," in *Proceedings of the IEEE On-Line Testing Symposium*, Rhodes, 2008, pp. 273-279.

[98] F. Corno, M. Sonza Reorda, G. Squillero, and G. Cumani, "Fully automatic test program generation for microprocessor cores," in *Design, Automation and Test in Europe*, 2003, pp. 1006-1011.

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

[100] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, United States: Morgan Kaufmann Publishers, 2006.

[101] Standard Performance Evaluation Corporation, "SPEC Benchmark," *SPEC Newsletter*, vol. 6, no. 3, Sep. 1994. [Online]. http://www.spec.org/cpu92/

[102] P. S. Ahuja, D. W. Clark, and A. Rogers, "The performance impact of

incomplete bypassing in processor pipelines," in *Proceedings of the International Symposium on Microarchitecture*, Ann Arbor, 1995, pp. 36-45.

[103] T. Y. Hsieh, M. A. Breuer, M. Annavaram, S. K. Gupta, and K. J. Lee, "Tolerance of Performance Degrading Faults for Effective Yield Improvement," in *Proceedings of the International Test Conference*, Austin, 2009, pp. 1-10.

[104] S. R. Makar and E. J. McCluskey, "On the Testing of Multiplexers," in *Proceedings of the International Test Conference*, Washington, 1988, pp. 669-679.

[105] C. Stroud, J. Sunwoo, S. Garimella, and J. Harris, "Built-in self-test for system-on-chip: a case study," in *Proceedings of the International Test Conference*, 2004, pp. 837-846.

[106] D. Gizopoulos, A. Paschalis, Y. Zorian, and M. Psarakis, "An effective BIST scheme for arithmetic logic units," in *Proceedings of the International Test Conference*, Washington, 1997, pp. 868-877.

[107] A. J. van de Goor, *Testing Semiconductor Memories, Theory and Practice*. New York, United States of America: John Wiley & Sons, 1991.

[108] R. Nair, "Comments on "An optimal algorithm fot testing suck-at faults in random access memories"," *Transactions on Computers*, vol. 28, no. 3, pp. 258-261, Mar. 1979.

[109] S. Ostendorff, H. D. Wuttke, J. Sachße, and S. Köhler, "A new approach for adaptive failure diagnostics based on emulation test," in *Proceeding on Design, Automation and Test in Europe*, Dresden, 2010, pp. 327-330.

[110] E. J. Marinissen, et al., "Adapting to adaptive testing," in *Proceedings on Design, Automation and Test in Europe*, Dresden, 2010, pp. 556-561.

[111] A. Khoche, D. Chindamo, M. Braun, and M. Fischer, "Selective and accurate fail data capture in compression environment for volume diagnostics," in *Proceedings of the International Test Conference*, Santa Clara, 2006, pp. 1-10.

[112] NXP. (2012, Oct.) UM10204-I2C-bus specification and user manual. [Online]. http://www.nxp.com/documents/user_manual/UM10204.pdf

[113] F. Karimi, Z. Navabi, W. M. Meleis, and F. Lombradi, "Using data compression in automatic test equipment for system-on-chip testing," *IEEE Transactions on Instrumentation and Measurement*, vol. 53, no. 2, pp. 308-317, Apr. 2004.

[114] S. Hamdioui, A. J. van de Goor, and M. Rodgers, "March SS: A Test for All Static Simple RAM Faults," in *Proceedings of the International Workshop on Memory Technology, Design and Testing*, 2002, pp. 95-100.

[115] T. M. Chin, L. L. Chung, and C. W. Wen, "A self-diagnostic BIST memory

design scheme," in *Records of the International Workshop on Memory Technology, Design and Testing*, San Jose, 1994, pp. 7-9.

[116] M. de Carvalho, et al., "Optimized embedded memory diagnosis," in *Proceedings on the International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, Cottbus, 2011, pp. 347-352.

[117] C. Selva, et al., "A Programmable Built-In Self-Diagnosis for Embedded SRAM," in *Records of the International Workshop on Memory Technology, Design and Testing*, 2004, pp. 84-89.

[118] L. M. Ciganda, F. Abate, P. Bernardi, M. Bruno, and M. Sonza Reorda, "An enhanced FPGA-based Low-cost Tester Platform exploiting effective Test Data Compression for SoCs," in *Proceeding of the International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, Liberec, 2009, pp. 258-263.

[119] D. Appello, et al., "Exploiting programmable BIST for the diagnosis of embedded memory cores," in *Proceedings of the International Test Conference*, Austin, 2003, pp. 379-385.

[120] Xilinx. (2005, Mar.) XUP Virtex II Pro Development System HW Reference Manual. [Online]. http://www.xilinx.com/univ/XUPV2P/Documentation/XUPV2P_User_Guide.pdf

[121] Xilinx. (2011, Jul.) Virtex II Pro Platform FPGAs Complete Datasheet. [Online]. http://www.xilinx.com/support/documentation/data_sheets/ds136.pdf

[122] A. B. Chatfield, *Fundamentals of High Accuracy Inertial Navigation*, I. American Institute of Aeronautics and Astronautics, Ed. 1997.

[123] IEEE, "1293-1998/Cor 1-2008 - IEEE Standard Specification Format Guide and Test Procedure for Linear, Single-Axis, Nongyroscopic Accelerometers," IEEE Standard E-ISBN 978-0-7381-6869-2, 2008.

[124] IEEE, "517-1974 - IEEE Standard Specification Format Guide and Test Procedure for Single-Degree-Of-Freedom Rate-Integrating Gyros," IEEE Aerospace and Electronic Systems Society E-ISBN 0-7381-0544-9, 2002.

[125] STMicroelectronics, "LIS331DL Datasheet - MEMS motion sensor: 3-axis - ±2g/±8g smart digital output "nano" accelerometer," STMicroelectronics Datasheet, 2008.

[126] STMicroelectronics, "L3G4200D Datasheet - MEMS motion sensor: ultra-stable three-axis digital output gyroscope," STMicroelectronics Datasheet, 2010.

[127] H. Wang. (2009) Synopsis. [Online]. http://www.synopsys.com.cn/information/snug/2009/test-point-insertion-for-test-coverage-improvement-in-dft

[128] A. Krstic, W. C. Lai, K. T. Cheng, L. Chen, and S. Dey, "Embedded Software-Based Self-Test for Programmable Core-Based Designs," *IEEE Design & Test of Computers*, vol. 19, no. 4, pp. 18-27, Aug. 2002.