G. Squillero, M. Rebaudengo

*Test Techniques for Systems-on-a-Chip*

December 2005
Fast innovation in VLSI technologies makes possible the integration of a complete system into a single chip (System-on-Chip, or SoC). In order to handle the resulting design complexity, reusable modules (cores) are being used in many SoC applications. System designers can purchase cores from core vendors and integrate them with their own User-Defined Logic (UDL) to implement SoCs. Core-based SoCs show important advantages: the cost of the end-product is decreased, and thanks to design re-use, the time-to-market can be greatly reduced.

The manufacturing test of such systems is a major challenge for industries as well as for the research community. The related issues can be layered as follows: 1) core layer: each core embedded into the SoC asks for an accurate test procedure, allowing the extraction of those information required to individuate the causes of technology weakness, 2) system layer: a quick and allowed overall SoC test plan has to be defined, 3) test application layer: the description of the test plan has to be easily produced in a convenient language in order to be read and executed by the selected Automatic Test Equipment (ATE).

In general, commonly employed industry practices are based on ad-hoc solutions approaching each single layer separately. Powerful core layer test structures often exploit particular bus structures suitably devised to fit the SoC architecture, but hardly reusable in a different design. Moreover, such an approach does not allow for a standardized test program generation, but relies on the product engineer’s ability to merge the test program application for each core while considering the ATE requirements and constraints. Therefore, the manufacturing test flow becomes a time-consuming job often requiring significant efforts in terms of human resources.

The proposed document addresses standardization of the SoCs manufacturing test resorting to a set of structures and techniques compliant with the IEEE 1500 Standard for Embedded Core Test (SECT). The set of guidelines proposed in this work are aimed at defining a flexible manufacturing test flow answering the
industrial needs for easing the interoperability within the itemized layers and reducing the required development timings for the complete test plan.

Regarding the core layer issues, the contribution of this work consists in a set of flexible Infrastructure IPs (I-IPs) aiming at the advanced test and diagnosis of memory, processor and logic cores. These I-IPs exploit a high degree of programmability allowing the customization of the applied tests and their reuse in different stages of the production flow. In details, programmable I-IPs have been designed for the test and the diagnosis of Flash and SRAM modules, for supporting software-based self-test and diagnosis of processor and logic cores. All the introduced structures integrate a diagnosis-oriented IEEE 1500 wrapper able to manage multiple test frequencies; moreover, they own a common data communication protocol, thus simplifying their insertion and adoption in industrial frameworks.

Concerning the system layer, a diagnosis-oriented Test Access Mechanism (TAM) is described in this document. Such a system layer structure suits for SoCs including the aforementioned core layer I-IPs, since it is able to efficiently manage the test procedure re-execution under different parameters and constraints often required for diagnosis. Moreover, a suitable micro-programmable I-IP has been designed based on the communication protocol defined in the core layer and it is able to manage the diagnosis with a notable advantage in test application time.

Concerning the application layer, a software tool able to automatically generate the STIL/CTL test description starting from the knowledge of the core and system layer characteristics has been proposed and practically evaluated. This tool elaborates a provided test scheduling: it verifies its validity with respect to the set of ATE constraints; it computes the measure of the test time; finally, it provides the comprehensive test description exploiting a set of STIL/CTL language macros implementing the defined IEEE 1500 based test data communication protocol.

The capabilities and low cost of the proposed structures for SoC test and diagnosis during manufacturing test have been experimentally demonstrated; currently, R&D divisions of STMicroelectronics are employing such test techniques for tuning the production flow and for populating their yield-
improvement databases. Additionally, many papers have been published in international test events. The advantages of the proposed methodology involve both the necessary test and diagnosis application time; moreover, the set of given guidelines introduces a not negligible reduction in terms of human resources needed, allowing the automation of the manufacturing flow based on an effective IEEE standard.

The project was supported by EU's ALFA Programme.
Summary

Chapter 1. Introduction.......................................................................... 1

Chapter 2. Systems-on-a-Chip manufacturing test......................... 6
  2.1 Core layer state-of-the-art test solutions ............................. 9
    2.1.1 Memory core test architectures ................................. 10
    2.1.2 Processor core test architectures ............................. 13
    2.1.3 User-Defined Logic cores test architectures .............. 15
  2.2 System layer state-of-the-art test solutions ....................... 16
  2.3 Application layer state-of-the-art test solutions ................ 19

Chapter 3. The proposed IEEE 1500 SECT compliant test flow ...... 21
  3.1 Core layer .......................................................................... 22
    3.1.1 Memory cores ......................................................... 23
    3.1.2 Processor cores ....................................................... 38
    3.1.3 User-Defined Logic cores ....................................... 49
  3.2 System layer ...................................................................... 62
    3.2.1 Diagnosis requirements at the system layer ............. 63
    3.2.2 The diagnosis-oriented TAM ................................... 65
    3.2.3 The I-IP structure ..................................................... 68
    3.2.4 Experimental evaluation .......................................... 71
3.3 Application level ................................................................. 74
  3.3.1 Test structures and strategy description ....................... 76
  3.3.2 Verification of a test plan .................................................. 79
  3.3.3 Evaluation of the test time ............................................. 79
  3.3.4 Test program generation ............................................... 80
  3.3.5 Experimental evaluation ............................................. 81
Chapter 4. Final remarks ...................................................... 88
Appendix A. A tool for Fault Diagnosis and data storage minimization .............................................................. 90
Appendix B. An industrial case-of-study ............................... 108
References ............................................................................. 122
Chapter 1. Introduction

In the semiconductor industry, the manufacturing phase deals with the increase of the yield for devices realized in an emerging technology. During the manufacturing phase a large amount of information about the failures affecting the build devices is retrieved and used to

- characterize the inspected technology by defining its capabilities and constructing limitations
- define a set of Design-for-Manufacturing rules suitable to increase the technology quality as soon as possible
- tune the industrial process in order to avoid recurrent constructive defects.

By referring to the typical semiconductor yield trend, the graph reported in figure 1 clarifies this concept: it shows how the yield figure is very low when starting the development of a new technology (that means to use a scaling factors higher than the consolidated or to adopt a device organization different from the usual), and it slowly grows until an acceptable quality level has been reached. The fastest is this growth, the shortest is the time-to-market.

Fast innovation in VLSI technologies makes possible to integrate a complete system into a single chip (System-on-Chip, or SoC). In order to handle the resulting design
complexity, reusable cores are being used in many SoC applications. System designers can purchase cores from core vendors and integrate them with their own User-Defined Logic (UDL) to implement SoCs. Core-based SoCs show important advantages: the cost of the end-product is decreased, and thanks to design re-use, the time-to-market is greatly reduced. A typical SoC structure is shown in figure 2.

![Diagram showing yield evolution by elapsed time.](image)

**Fig. 1:** the yield evolution by elapsed time.

The manufacturing test of such systems is a major challenge for industries as well as for the research community and its issues can be divided in three topic layers: 1) core layer: each core embedded into the SoC asks for an accurate test procedure, allowing the extraction of those information required to deeply investigate the causes of technology weakness, 2) system layer: a quick and cheap overall SoC test plan have to be defined, 3) application layer: the description of such test plan have to be easily produced in a convenient language in order to be read and executed by the selected Automatic Test Equipment (ATE).
In general, commonly employed industry practices are based on ad-hoc solutions approaching each single layer issue separately. Powerful core layer test structures are often connected using particular bus structures suitably thought to fit investigated SoC structure, but rather reusable in a different design. Moreover, such an approach does not allow for a standardized test program generation, but relies on the product engineers ability to merge the test program application for each test while taking care of the ATE requirements and constraints. Therefore, the manufacturing test flow often requires several efforts in terms of human resources and it becomes a time-consuming job.

The proposed document addresses standardization of the SoC’s manufacturing test resorting to a set of structures and techniques compliant with the IEEE 1500 SECT. The set of guidelines proposed in this work are aimed at defining a flexible manufacturing test flow answering the industrial asks for easing the interoperability within the itemized layers and reducing the required timings.

Regarding the core level issues, the contribution of this work consists in a set of flexible Infrastructure IP (I-IP) aiming at the advanced diagnosis of memory, processor and logic cores. These I-IPs exploit a high degree of programmability allowing the customization of the applied test and allowing their reuse in different stages of the production flow. In detail, a programmable Built-in Self-Test engine has been designed
for the test of Flash and SRAM; an I-IP for supporting the Software-Based Self-Test of processor cores have been designed to ease the extraction of test and diagnosis data during the technology yield ramp-up; a programmable pseudo-random Built-in Self-Test has been designed for logic core. All the introduced structures have been surrounded with a diagnosis-oriented IEEE 1500 wrapper and a shared data communication protocol has been formalized, thus simplifying their insertion in industrial frameworks. Their capability and low cost have been experimentally demonstrated; currently, R&D divisions of STMicroelectronics adopt their usage for tuning the production flow and for populating their yield-improvement databases.

For the system level, a diagnosis-oriented Test Access Mechanism (TAM) is described in this work. Such a system layer structure suits for SoCs including the aforementioned core layer I-IPs, since it is able to efficiently manage the test procedure re-executions under different parameters and constraints required for diagnosis. Moreover, a suitable micro-programmable I-IP has been designed based on the defined communication protocol defined in the core layer; this I-IP is able to manage the diagnosis operations usually demanded to the ATE with a notable computation time gain.

Concerning the application layer, a software tool able to automatically generates the STIL/CTL test description starting from the knowledge of the core and layer characteristics has been realized. This tool elaborates a provided test scheduling: it verifies its validity with respect to the set of ATE constraints; it computes the measure of the test time; finally it provides the compliant test description exploiting a set of STIL/CTL language macros implementing the defined test data communication protocol.

Manufacturing test issues for SoCs and the state-of-the-art solutions are introduced in chapter 2: this section is organized in an analytic manner, focusing on the problems involved in three defined test layers - core, system and application level -, and summarizes the most relevant solutions proposed by the industry and the academy. In chapter 3, the contribution given by this work in the field of SoCs manufacturing test is
detailed, describing a set of developed technique and demonstrating their effectiveness and efficiency on appropriate industrial case of studies. Finally, appendix A details of an algorithm allowing the fault classification starting from the information retrieved by using the structures and methods illustrated, and appendix B proposes an industrial case of study exploiting the most of the concepts included in this work.
Chapter 2. Systems-on-a-Chip manufacturing test

The relevant System-on-a-chip manufacturing test aspects may be divided into three layers, each one related to a particular step of the SoC design flow, as follows:

- the **Core layer**
- the **System layer**
- the **Application layer**.

Figure 3 shows a conceptual diagram of the introduced three-layer structure for SoC manufacturing test.

The **Core layer** is managed by the designer of the core itself; at this level, the test job often corresponds in selecting which of the available *Design-for-Testability* or *DfT* techniques is the most convenient, and consequently in generating a set of *test patterns* to be applied to obtain a required level of fault coverage and, eventually, allowing the fault diagnosis of the core. Commonly adopted DfT techniques resort to the insertion of additional hardware like scan-chains, Built-In Self-Test or BIST architectures or special purpose circuitries providing more controllability and/or accessibility to the core design, such as Cyclic Redundancy Code (CRC) registers. Possible synergies between cores included in the SoC can be exploited, for example by requesting a core to test another
Chapter 2. Systems-on-a-Chip manufacturing test

one and this choice depends on the core type and on the SoC overall structure, when its final composition is just known.

In general, the insertion of a suitable BIST engine is the most desirable solution, if applicable. This approach is based on the integration of pattern generator and response evaluator state machines within the core under test; BIST adoption increases the autonomy of the core test and allows for at-speed testing, therefore increasing the fault coverage. In literature, hardware structures inserted for test application exclusively are often referred as Infrastructure IP (I-IP), since they provide the SoC any additional functionality but the support for test and/or diagnosis [1].

The System layer is managed by the system integrator. During this development phase, the final structure of the SoC is completely known, and the included cores are connected in the final design of the SoC structure. System integrators have to face two critical test aspects:

- the limited number of external pins and, consequently, the reduced accessibility of the cores embedded in SoCs raise the need for a Test Access Mechanism or TAM architecture allowing test inputs to reach the core and test response to be read from the top level of chip; the trade-off is between the allowed bandwidth and the number of additional test resources required such as the number of dedicated pins and routing resources [2]

- a time convenient core test execution order, also called Test Scheduling, has to be defined, taking into consideration the limitation imposed by both the TAM architecture employed and the SoC technology limitation [3].

Very efficient approaches provide solutions able to minimize at the same time the costs for the TAM and the Test Scheduling application time: such solutions are usually based on the use of Test Interfaces surrounding each core – or group of cores – and providing a homogeneous way to communicate test data.
Moreover, the effectiveness of employing hardware structures at this test layer has been widely demonstrated and their usage formalized to achieve complete scalability and reusability.

![Diagram showing the three layers of SoCs manufacturing test](image)

**Fig. 3:** the three layers of SoCs manufacturing test.

The **Application layer** is finally addressed by product engineers when the resulting device has to be tested by a given Automated Test Equipment or ATE, and basically deals with the test *waveforms* management, sequenced by a given Test Scheduling. ATEs are sophisticated and expensive machines in care of the physical application of electrical stimuli to the devices under test pins and of the circuit responses evaluation. The test program including the patterns needs to be fed into ATEs resorting to an appropriate *test description language*, which in the past was strictly manufacturer-dependent [4]. Several complications may be introduced by this layer toward the complete automation of the flow, since it deals directly with the ATE hardware and software specific structures. The multitude of commercially available ATE’s and the significant architectural differences they show, requires a conscious partitioning of the flow and tools to take into account ATE dependencies, without penalizing the flow cleanliness.
Current industrial trends are pushing toward solutions looking at the SoC testing problem from a higher point of view; such solutions take into consideration the requirements coming from all of the three defined layers. Figure 4 graphically shows the SoC manufacturing test hardware and software progressive implementation for each layer.

Fig. 4: Conceptual view of a generic industrial test plan

2.1 Core layer state-of-the-art test solutions
SoCs can be composed of many different cores. In this paragraph, a coarse categorization has been introduced in order to provide the reader an analytic overview of the core test layer commonly employed techniques.

The cores structures considered are included the following categories:

- Memory cores
- Processor cores
User Defined Cores.

### 2.1.1 Memory core test architectures

Nowadays, embedded memory cores often determine the yield in production processes of Systems-on-Chip (SoCs), as they tend to consume most of the transistors in SoCs and their density is continuously rising, as shown in figure 5.

![Graph showing memory occupation in SoCs trend until 2015, shown in percentage](image)

**Fig. 5:** the memory occupation in SoCs trend until 2015, shown in percentage

Extensive research on fault detection in embedded memories has been performed and efficient algorithms have been proposed and implemented [1][5]. BIST provides an effective way to automatically generate test sequences, compressing the outputs and evaluating the goodness of embedded memory cores, and BIST-based solutions are now very popular [6][7]. The typical memory BIST implements a March algorithm [8] composed of a sequence of March elements, each corresponding to a series of read/write operations on the whole memory. Different hardware approaches have been proposed in the literature in order to implement BIST-based March test algorithms.

The hardwired BIST approach is the most widely used. It consists in adding a custom circuitry to each core, implementing a suitable BIST algorithm [9]. The main advantage of this approach is that the test application time is short and the area overhead is relatively small. Hardwired BIST is also a good way to protect the intellectual property
Chapter 2. Systems-on-a-Chip manufacturing test

contained in the core: the memory core provider needs only to deliver the BIST activation and response commands for testing the core without disclosing its internal design. At the same time, this approach provides very low flexibility: any modification to the test algorithm requires redesigning the BIST circuitry.

![Diagram](image)

*Fig. 6: a), b) and c) show a conceptual view of the hardwired, soft and programmable BIST structure, respectively*

The soft BIST approach [10] assumes that a processor is already available in the SOC: the processor is exploited to run a program performing the test of the other cores. The test program executed by the processor applies test patterns to each core under test and checks for the results. The test program is stored in a memory containing also the test patterns. This approach uses the system bus for applying test patterns and reading test responses, and it guarantees a very low area overhead, limited to the chip-level test infrastructure. The disadvantage of this approach is mainly related to the strict dependence of the test program on the available processor. As a result, the core vendor needs to develop for the same core different test programs, one for each processor family, thus increasing the test development costs. Moreover, intellectual property is not well protected, as the core vendor supplies to the user the test program for the core under test. Finally, this approach can be applied only to cores directly connected to the system bus; the approach cannot be applied if the core is not completely controllable and observable.
Chapter 2. Systems-on-a-Chip manufacturing test

An alternative approach is that one usually denoted as *programmable BIST* [11]. The core vendor develops a DFT logic, which wraps the core under test and includes a custom processor, which is exclusively devoted to test the core. The advantages of this architecture are manifold: the intellectual property can be protected, only one test program has to be developed, and the design cost for the test is very reduced; the technique provides high flexibility since any modification of the algorithm simply requires a change in the test program; the test application time can be taken under control thanks to the efficiency of the custom test processor, and the test can consequently be executed at-speed. Finally, each core is autonomous even from the test point of view, and its test only requires activating the test procedure and reading the results, as for hardwired BIST. The main potential disadvantage is the area overhead introduced by replicating the custom processor in each core under test. However, due to the very limited size of the processor, this problem is marginal (especially when applied to cores including medium- and large-sized memories) and can be completely overcome when sharing the BIST circuitries among many memory cores.

During the manufacturing yield ramp up process, more details about the arising faults are needed, beyond those given by test solutions producing only go/nogo information. A commonly employed solution consists in adopting diagnosis-oriented march test algorithms [12] and in building the so-called *failure bitmap* [13][14], which keeps tracks of every fault detection encountered during the test execution. Several BIST structures proposed in the literature [14][15] [16][17] and adopted by industries include diagnostic features, allowing to obtain precise data about physical failures affecting memory cores. The cost for diagnosis can be very high due to the following causes:

- the time needed to collect failure signatures can be very high dependently from faulty scenarios
- additional circuitries (and pins) have to be integrated into the original BIST design
• the tasks to be performed by test equipments (ATE) to support diagnosis are
  often rather expensive in terms of pattern occupation, flexibility in the execution
  of parametric flows, and access description interoperability.

Several papers proposed compression approaches in order to minimize these
limitations. A Huffman-based compression method is proposed in [18][19][20] where
each faulty word is compressed and extracted. A suitable approach to reduce the time for
extraction and the tester storage is also presented in [21]: the failure bitmap is first
compressed using a vertical and horizontal algorithm, then, once extracted, it is
decompressed and analyzed by a program running on the tester equipment.

\subsection{2.1.2 Processor core test architectures}

Today, almost every SoC includes at least one microprocessor or microcontroller
core, which may be a general or a special purpose processor, surrounded by different
memory cores of various size used for code and data storage. Unfortunately, the
complexity of SoCs including deeply embedded cores often makes their testing very
hard.

Self-test techniques are expected to play a key role into this scenario [10]. Self-test
approaches can guarantee high fault coverage by autonomously performing the test at the
nominal frequency of the IC (at-speed test) and drastically reduce the cost of the required
Automatic Test Equipment (ATE). Self-test approaches are divided into two categories:

\begin{itemize}
  \item Hardware-based Self-test
  \item Software-based Self-test.
\end{itemize}

Hardware-based Self-test architectures require additional hardware structures (e.g.,
Built-in Self-test, Logic Built-in Self-test, etc.). The adoption of these techniques is
particularly suited to IP cores not strictly constrained in terms of timing and consumption
[6]: hardware modifications to support Self-Test allow at-speed test and relieve the ATE
from test management. However, whereas processor cores are performance-constrained,
any introduced additional logic can fatally impact their efficiency and power
consumption, and, in general, solutions adopting logic BIST [22] or based on scan chain insertion might be unpractical also in terms of test application time due to the excessive time needed to upload scan patterns and read test results.

Software-based Self-test methodologies appear to better suit embedded processor cores test. Software-based strategies are based on the execution of suitably generated test programs [23]; therefore, no extra hardware is required, and the existing processor functionalities are used for its test. No modifications of the IP are needed and the performance is not decreased, since the test is performed at the operative speed of the embedded core. In the past, several efforts were made to devise effective techniques for generating test programs able to obtain high fault coverage figures at acceptable costs: recently, some significant results were achieved in this field even when pipelined processors are considered [24][25].

However, some requirements regarding the test environment indispensable for such test solutions should be carefully considered:

- a memory module available on the SoC for storing the test program and a mechanism to upload the code should be provided
- a method to start the execution of the self-test program should be identified
- a procedure to monitor and extract test results should be defined.

The advantages stemming from the adoption of Self-test methodologies cannot be bounded to single core test considerations, but their convenience should be evaluated.
also in terms of the whole SoC test strategy. Reduced ATE control requirement and independence with respect to the test frequency must be considered in order to maximize the economy of the overall test environment; as a matter of fact, the test of more than one Self-testable embedded component can be concurrently performed [26]. To do that, each embedded IC requires being reachable from the top layer of the chip and test engineers have to take care in the power consumption bounds during the overall test of their SoCs.

If the localization of faults in processor cores is aimed, additional efforts are required in the test set generation: diagnostic set construction is a time-consuming activity [27][28]. Hard-to-test faults require a high computational effort for their coverage, but once detected they are usually easy to diagnose; easy-to-test ones, on the other hand, may be difficult to discriminate from each other and require a special effort for diagnosis. This difficulty can lead to long diagnostic tests, with correspondingly long application times and high costs.

2.1.3 User-Defined Logic cores test architectures

In SoCs development flow, system designers often purchase cores from core vendors, thus integrating them with their own User-Defined Logic (UDL). While the convenience of this SoC assembly strategy is well-known in terms of time-to-construct the final system, the test of the UDL core is difficult in the most of cases, due to their wide type range. Different approaches can be adopted for testing logic cores; they are normally grouped in the following classes:

- scan based
- synergy based
- pseudo-random based.

In scan based and logic BIST approaches, a set of patterns are generated using automatic tools (ATPGs) and applied to the circuit. In the sequential approach, the calculated patterns are sequentially sent to the circuit and responses read after each application and any additional internal structure is added in order to improve the
effectiveness of the patterns. On the contrary, in the scan approach, the controllability and the observability of the circuit are improved by modifying the common flip flop: the so-called scan cells allow writing and reading the content of the memory element during the test apply, and are connected to compose a scan chain. However, as a serial process is required to load and upload the scan chain, that approach requires onerous application time and heavy ATE requirements in terms of storage needed for test data and test application program.

The synergy test approaches \[29\][30] reuses of existing SoC functionalities for applying a decent test procedure. Some researchers \[31\][32][10] proposed to exploit an embedded processor to test the other components of the SoC: first the processor core is tested, and then a test program, executed by the embedded processor, is used to test the UDL cores. The use of embedded processors to test cores presents some advantages, such as that the test program (being in software) guarantees a high flexibility and the testing process can often be done at-speed. Moreover, the test process is completely executed inside the chip at its functional frequency, while the tester can work at a lower speed: this approach reduces the costs for the test equipment and guarantees high test quality. By the way, a mandatory condition for this approach to be applicable is that the embedded cores to be tested must be suitably connected to the processor, which in general is not always true.

An alternative and well documented technique is the pseudo-random pattern generation. Such approach is based on the Galois theories for the generation of pseudo-random number sequences starting from the definition of a characteristic polynom. Particular structures, called Autonomous Linear Feedback Shift Registers (ALFSR), are the hardware implementation for such kind of pattern generation strategy \[33\][34].

2.2 System layer state-of-the-art test solutions

The effectiveness of a test scheduling strategy heavily depends on the adopted test structure as highlighted by many previous works and can be evaluated in terms of core isolation and accessibility. As a matter of fact, the definition of test structures bounding
the core and test access mechanisms (TAMs) is a deeply investigated task. Currently, the most of the state-of-the-art solutions are compliant with the IEEE 1500 Standard for Embedded Core Test [35]: this standard defines test interface architectures, the so-called IEEE 1500 wrappers, which allow, besides flexibility and easy reuse, the usage in the application layer of high-level description in CTL language [4]. The schematic of an IEEE 1500 standard compliant wrapper is shown in figure 8.

![IEEE 1500 wrapper schematic](image)

**Fig. 8:** a generic structure of an IEEE 1500 standard wrapper [35].

Such test interfaces are connected by a test bus that constitutes the Test Access Mechanism (TAM) allowing reaching every core embedded in the SoC from its top level in order to apply the pattern set generated during the core layer phases. Other solutions propose centralized controllers, or a hierarchical distributed structure using hardware implementation of network protocols.

A very large number of test scheduling algorithm has been defined capable of minimizing the hardware requests in terms of test interface circuitry size and the time to apply stimuli when considering SoCs including lots of core equipped with scan chains. These approaches mainly address at the minimization of the TAM bus width resorting to opportune scan chain partition and bus signal assignment to included cores[2][29][37]: in
a particularly clever TAM architecture is defined called TestRail, while in another architecture named CAS-BUS guarantees flexibility, scalability and reconfigurability of the overall TAM scenario. Additionally, these test algorithms are thought to maximize the parallelization of the core test application without overcoming the SoC technology-driven constraints, which are mainly related to power consumption. In practical terms, system integrators have to take care of defining test scheduling algorithms not consuming more than a SoC technology dependent threshold current.

On the contrary, when considering SoCs including a large number of BIST circuitry, the test bandwidth requirements are usually very low as the stimuli are internally generated; the effort for test application exclusively consists in launching the self-test execution, waiting until the internal test is finished, and reading the results. Differently from scan chain based SoCs, the key problem is to define a global test strategy compliant with the different requirements imposed by the included BIST modules as underlined in [44]. An efficient test scheduling approach in these cases should be able to guarantee flexibility in terms of test structure and scheduling, reusability and independently from the BIST implementation and capabilities. Strategies for time minimization must take into account technology-driven constraints (i.e., power limits, wires length, etc.), while more complex Test Access Mechanisms (TAMs) are
mandatory to automate the extraction flow [29], and the storage space requested on ATE platforms becomes very large. In general, the most of the solutions provided in literature are based on the insertion of additional blocks able to manage the BIST scheduling, such as centralized controllers [45][46], distributed structures [47].

![Fig. 10: The HD-BIST distributed test structure proposed in [47].](image)

As far as SoC diagnosis is concerned, the requirements for an efficient access to cores equipped with BIST architectures must be reconsidered: in particular, as the occurrence of faults cannot be forecasted, external test equipments are requested to manage at runtime the entire diagnostic flow, considering in their computation the physical constraints. In addition, since the diagnostic procedures often require repeated accesses to the faulty cores, additional latency is introduced in programming BISTs. The weight of this latency depends in a minor part on the physical reprogramming of the used I-IPs, and substantially on the ATE environment adjustment.

### 2.3 Application layer state-of-the-art test solutions

Several academic and industrial solutions for test structure definition and optimization have been proposed involving the three individuated test layers. However, most of them deal with the management of homogeneous test structures, in most cases scan chains or replicated instances of the same BIST architecture.
Core and system layer solutions compliant with the IEEE 1500 Standard for Embedded Core Test [35] allows for the automatic generation of high-level description in CTL language [4]. This language is said to be “native”, since it can be read directly by the Automatic Test Equipment without any additional conversion. Another native ATE language is the STIL language [48] that provides the product engineer the ability to describe with macros recurring test procedures.

For complex systems, including several cores tested in various ways, the generation of the test programs in a selected language often becomes a very expensive task, despite the fact that it is commonly perceived as a trivial process. Planning for the test program up front and evaluating it at an early stage of the design-to-test cycle can significantly reduce time-to-market as well as overall cost, and improve product quality. The more information can be transferred from the design environment to the test phase, the easier the work is [49][50]. Test integration for systems including cores equipped with heterogeneous DfT structure and protocols, and the relative cost evaluation, are mostly done manually by skilled test engineers. Exploring the performance of various possible test configurations by switching from one solution to another is a time-consuming task for designers, and the job is further complicated by the constraints and requirements coming from the available ATE.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

In this chapter it is presented an IEEE 1500 SECT compliant manufacturing test flow for SoC. This flow takes into consideration the hardware and the software aspects introduced in section 2 and divided in:

- the Core layer
- the System layer
- the Application layer.

Regarding the core layer, a set of hardware architecture, also called Infrastructure IP (I-IP), are presented. These I-IPs enable the fault diagnosis for memory, processor and user defined logic cores and each of them equipped with an IEEE 1500 test interface or wrapper. For all the presented test structure, a software environment has been developed in order to effectively and easily control their abilities; the software tools implemented aims at fully managing the test structures and analyzing the retrieved results. All the proposed hardware structures are suitably thought to enable the core diagnosis; the principles of a tool able to perform the diagnostic fault classification are detailed in appendix A.
A diagnosis oriented TAM mechanism exploiting the IEEE 1500 compliancy of the adopted core layer structure is proposed concerning the SoC system layer of the manufacturing test topic. Moreover, a particular Infrastructure IP module is presented able to autonomously manage the diagnosis processes for groups of cores equipped with the developed core layer I-IPs.

Finally, a software platform is described for answering some of the test application layer demands. This platform is able to generate the overall description of the SoC test in the standardized IEEE 1450 STIL language, accordingly with the characteristics of each core test type, with the SoC system layer test structure and scheduling, and with the characteristics of the ATE. Additionally, this tool is able to precisely estimate the cost for the SoC complete test in term of time required.

3.1 Core layer

The core layer test architecture proposed in the following paragraphs has been suitably designed for the test and diagnosis of

- Memory cores
- Processor cores
- User Defined Logic cores.

All the defined structures answer to industrial requests and share a test access method based on the IEEE 1500 SECT. In such a way that every designed I-IP could provide a common test access strategy, they always include two ad-hoc ports:

- an instruction port, receiving high level commands controlling the structure behavior
- a data port, used as an input or output port depending on the high level command received on the instruction port.

The common test access structure and the IEEE 1500 SECT compliant wrapper surrounding each core are shown in figure 11.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

3.1.1 Memory cores

A custom March based programmable BIST able to execute a test program for SRAM and Flash memory cores is described in this section. In the SoC manufacturing test contest, the contribution of this work with respect to the approached topic is twofold: it provides the description of a programmable architecture oriented to diagnosis and it delineates the guidelines for including test access structures and define high level test application protocols.

The proposed programmable-BIST conceptual test architecture is shown in figure 12: the march based processor fetches the code to be decoded and executed from a code memory area (usually a SRAM); the march stimuli sequence described in the code memory is applied to the embedded memory core; the programmable BIST is finally designed to be fully controllable by an external interface compliant with the IEEE 1149.1 and 1500 test standards, which allows the ATE to manage the test by sending high level instructions.
The proposed IEEE 1500 SECT compliant test flow

Chapter 3. The proposed IEEE 1500 SECT compliant test flow

The programmable BIST internal structure

The internal architecture of the March based programmable BIST is divided into 2 functional blocks, as shown in figure 13: a Control Unit to manage the test algorithm and a Memory Adapter to apply it to a specific memory core. Splitting the processor in two parts allows reducing the cost for its extension to new cores.

Fig. 13: Processor internal architecture.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

The **Control Unit** manages the test program execution; it receives management commands from the ATE (for example START and RESET commands), and fetches/decodes the instruction read from the micro-code memory. Therefore, he generates the control signals that the Memory Adapter executes on the memory core under test. The Control Unit includes an Instruction Register (IR) and a Program Counter (PC). By means of control commands, the Control Unit allows the correct update of some registers located in the Memory Adapter and devoted to customize the test and diagnosis procedures. This choice simplifies the processor reuse in different applications without the need for any re-design, e.g., the execution of a different test program, the test of a memory with a different size or the test of different memory models.

The **Memory Adapter** includes all the test and diagnosis registers used to customize and correctly execute the March algorithm:

- the Control Address register (**Current_address**): it contains the address of the currently accessed memory cell
- the Control Memory registers:
  - **Current_data**: it contains the data to be written into the memory during the current read/write operation
  - **Received_data**: it contains the data read from the memory
- the Control Test registers:
  - **Dbg_index**: it contains the index to access to the databackground register file
  - **Step**: it contains the number of steps to be executed. The size of this register is \( \log_2 M \), being \( M \) the total number of read/write operations executed by the March algorithm
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

- **Direction flag**: a bit specifies the direction (forward or backward) of the March Element
- **Timer**: it contains the number of waiting clock cycles in order to introduce pauses into the test algorithm execution

- **the Result registers**:
  - **Status (Status Register)**: it contains 2 bits (E and S); E is active when the March algorithm has reached its end, S is active when the BIST algorithm reaches the number of steps to be executed
  - **Err (Error Register)**: it counts the number of times a fault is detected in a cell (i.e., when at least a bit in the cell is faulty); its size is equal to $\log_2 M$ bits
  - **Result (Result Register)**: it contains the information concerning the last detected fault. The stored data are:
    - **STEP**, the ordinal number of the operation which has just been executed, its size is $\log_2 M$ bits
    - **DATA**, the logical xor between the read and the expected words; its size is $n$ bits, being $n$ the parallelism of the memory.

The Memory Adapter is also composed of a set of registers containing constant values defined at the design step according to the characteristics of the memory under test and to the test algorithm:

- **Add_Max** and **Add_Min**: the first and the last addresses of the memory under test, respectively
- **DataBackGround**: it contains the databackground set of values in use during the test cycle
- **Dbg_max**: it contains the reference to the databackground value in use.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

The functional modes of the processor are the following:

- **normal**, the processor is inactive; this is the default mode during the system normal mode
- **reset**: it is entered when the `RESET 1500` instruction is sent; this instruction activates the processor, so that it becomes active and ready to run the program
- **run**: it is entered when the `RUNBIST 1500` instruction is sent, which forces the processor to start the program execution.

![Memory Adapter generic internal architecture](image)

**Fig. 14:** Memory Adapter generic internal architecture.

The instruction set has been designed to support the widest range of March algorithms and to guarantee high flexibility and adaptability to the processor. Detailed information about the set of instructions supported by the processor are reported in table I.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

Tab. I: Processor Instruction Set.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_ADD</td>
<td>Current_address ← Add_Max</td>
</tr>
<tr>
<td></td>
<td>Direction flag ← BACKWARD</td>
</tr>
<tr>
<td>RST_ADD</td>
<td>Current_address ← Add_Min</td>
</tr>
<tr>
<td></td>
<td>Direction flag ← FORWARD</td>
</tr>
<tr>
<td>STORE_DBG</td>
<td>Current_data ← DataBackGround [Dbg_index]</td>
</tr>
<tr>
<td></td>
<td>Dbg_index ← Dbg_index + 1</td>
</tr>
<tr>
<td>INV_DBG</td>
<td>Current_data ← NOT (Current_data)</td>
</tr>
<tr>
<td>READ</td>
<td>Current_data ← Memory[Current_address]</td>
</tr>
<tr>
<td>WRITE</td>
<td>Memory[Current_address] ← Current_data</td>
</tr>
<tr>
<td>BNE Offset</td>
<td>if (Direction flag = BACKWARD) then</td>
</tr>
<tr>
<td></td>
<td>{ if (Current_address &lt;&gt; Add_Min) then</td>
</tr>
<tr>
<td></td>
<td>{ Program Counter ← Program Counter - Offset</td>
</tr>
<tr>
<td></td>
<td>Current_address ← Current_address - 1</td>
</tr>
<tr>
<td></td>
<td>} else Program Counter = Program Counter + 1</td>
</tr>
<tr>
<td></td>
<td>} else if (Current_address &lt;&gt; Add_Max) then</td>
</tr>
<tr>
<td></td>
<td>{ ProgramCounter ← ProgramCounter - Offset</td>
</tr>
<tr>
<td></td>
<td>Current_address ← Current_address + 1</td>
</tr>
<tr>
<td></td>
<td>} else Program Counter ← Program Counter + 1</td>
</tr>
<tr>
<td>LOOP Offset</td>
<td>Dbg_index = Dbg_index + 1</td>
</tr>
<tr>
<td></td>
<td>if (Dbg_index &lt; Dbg_max) then</td>
</tr>
<tr>
<td></td>
<td>Program Counter = Program Counter – Offset</td>
</tr>
<tr>
<td></td>
<td>else Program Counter + Program Counter + 1</td>
</tr>
<tr>
<td>SET_TIME data</td>
<td>Timer ← data</td>
</tr>
<tr>
<td>PAUSE</td>
<td>Timer = Timer - 1</td>
</tr>
<tr>
<td></td>
<td>if (Timer = 0) then</td>
</tr>
<tr>
<td></td>
<td>Program Counter = Program Counter + 1</td>
</tr>
<tr>
<td>END_CODE</td>
<td>Functional Mode ← Normal</td>
</tr>
</tbody>
</table>
The presented structure exactly suits for volatile embedded memories such as SRAM or cache, but requires some modification for other types of memory cores, such as Flash memory cores. In this particular case, a fitful architecture is shown in figure 15.

![Programmable BIST for Flash memory internal architecture.](image)

The Flash Manager needs to be customized to the flash memory under test, due to its strict dependence on the memory model used. It manages the specific control and access timing signals of the memory, while its behavior is controlled by the Memory Adapter. This protocol is suitable for using two distinct clock signals: the external clock connected to the Control Unit and Memory Adapter and an internal clock connected to the Flash Manager. While the external clock is provided by the SOC and can present higher frequency in order to speed up the program execution, the internal clock is used during memory access operations and must be chosen to properly satisfy the timing requirements of the memory under test. This approach increases the design flexibility enabling the execution of the test operation at a frequency not dependent on the memory model.

Beyond the additional required Flash manager module, the architecture of Programmable BIST architecture requires some modification in the Memory Adapter module. In particular, this module has to be able to execute suitable accesses to the Flash
memory core under test. Table II shows the modified Instruction Set of the Programmable BIST for Flash allowing the application of Flash memory oriented march tests described in [51][52][53].

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERASE_FLASH</td>
<td>Full memory array electrical reset</td>
</tr>
<tr>
<td>READ_FLASH</td>
<td>Current_data ⇐ Memory[Current_address]</td>
</tr>
<tr>
<td>PROGRAM_FLASH</td>
<td>Memory[Current_address] ⇐ Current_data</td>
</tr>
</tbody>
</table>

The IEEE 1500 wrapper

The wrapper, shown in figure 16, contains the necessary circuitry to interface the processor with the outside in a 1500 compliant fashion, supporting the commands for running the BIST and accessing to its results. The wrapper is compliant with the suggestions of the 1500 standardization group.

In addition to the mandatory components, the wrapper architecture includes the following Wrapper Data registers:

- **Wrapper Control Data Register (WCDR):** through this register the TAP controller sends the commands to the processor (e.g., the processor reset, the test program start, the result registers read, etc.)

- **Wrapper Data Register (WDR):** it is an I/O buffer register. The TAP Controller can read the diagnostic information stored in the result registers (*Status, Err* and *Result*). According to the command written in WCDR the outside world may execute one of the following operations involving WDR:

  - read from WDR the diagnostic information (i.e., the number of detected errors, the step corresponding to the last detected error and the faulty word) stored into the result registers
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

- write in WDR the number of steps to be executed by the March algorithm to be written into the Step register.

![Diagram](image.png)

**Fig. 16:** The proposed Wrapper Architecture.

The IEEE 1149.1 compliant TAP Controller plays the role of interfacing the ATE with the wrapper, and hence with the test module. The TAP Controller supports the following instructions (beyond the standard ones, such as BYPASS):

- **RESET:** puts a core into the reset state;
- **RUNBIST:** executes a complete run of the March program;
- **LOADSTEPS:** loads the number of operations to be executed by the March algorithm into the Step register; by default this number is equal to the number of operations required by a complete execution of the adopted March algorithm;
- **READSTATUS:** reads the Status register and verifies whether the March algorithm finished its task (either because it reached its end, or because it executed the specified number of operations);
- **READRESULT:** reads the Result register containing the information about the last error detected by the March algorithm.
**Chapter 3. The proposed IEEE 1500 SECT compliant test flow**

- **READERROR**: reads the $Err$ register containing the number of errors detected by the March algorithm.

*The Test and Diagnosis Program*

The developed ATE software module performs a test or a diagnosis session. According to the scheme shown in figure 17, the ATE is composed of 2 modules: a Test Program and a Bitmap Generator. The Test Program is in charge of controlling the test and diagnosis execution. It is composed of 2 tasks: a Run BIST task and a Response Analysis one. The Bitmap Generator program represents a higher-level procedure in charge of identifying fault(s) based on the collected information e.g., by exploiting the approach described in [54].

![Fig. 17: ATE General Software Architecture.](image_url)
When test is the only goal, using the retrieved information about the number of detected errors one can separate fault-free from faulty chips, and the process ends. In the case of testing:

- the Run BIST task sends the \texttt{RESET} instruction, which initializes the processor in the core and the \texttt{RUNBIST} instruction, which starts the execution of the test program.
- the Response Analysis task waits until the completion of the test program by polling the status of the BIST module using the \texttt{READSTATUS} instruction and accesses the result through the \texttt{READERROR} instruction.

When diagnosis is the concern, the ATE gathers all the information about each error detected by the test program on the faulty embedded memory. The ATE attains this goal by executing the following operations also shown in figure 18:

1. the Run BIST task sends the \texttt{RESET} and \texttt{RUNBIST} instructions as before.
2. the Response Analysis task executes the procedure:
   - it waits until the completion of the test program by polling the status of the BIST module using the \texttt{READSTATUS} instruction
   - it accesses the result through the \texttt{READERROR} instruction.
   - the information about the last detected error is retrieved through \texttt{READRESULT}.
   - the Bitmap Generator receives the information stored into the \texttt{Result} register and computes the address of the faulty memory cells and their faulty bit cells, updating the memory bitmap
3. The Run BIST task executes the \texttt{RESET} instruction and the \texttt{LOADSTEPS} instruction updating the number of operations to be executed by the March algorithm according to the data stored into the STEP field of the \texttt{Result} Register;
the program execution is again launched through the **RUNBIST** instruction.

4. Steps 2 and 3 are iteratively repeated for all the detected errors, and the related information are then extracted. Thanks to the algorithm adopted in the Bitmap Generator module the diagnosis process normally reaches its goal (i.e., identifying the fault type and location) before extracting the information about all the detected errors.

```plaintext
error_detected = True;
Step = Full;
while (error_detected)
{
    for (i=0 ; i<Step ; i++)
        March(i);
    Read (Status)
    If (Status <> 0)
    {
        read (Err, Result);
        Step = Result.step – 1;
    }
    else
        error_detected = False;
}
```

Fig. 18: Response Analysis task basic procedure.

The above procedure guarantees that faults can be identified with the maximum diagnostic capability allowed by the March algorithm implemented by the embedded processor. This means that all faults that could be diagnosed by the adopted March algorithm when directly implemented by an ATE having full access to the memory are still diagnosable by the described architecture.

The Programmable BIST approach allows to easily adapting the test algorithm to the specific test requirements.

The Instruction Set allows executing all the possible March Test algorithms. The instructions **SETTIME** and **PAUSE** can be inserted into the test program to detect data retention faults as reported in the example shown in table III.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

Different databackground (0,1, checkerboard, etc.) can be adopted: a suitable set of values into the DataBackGround register file have to be defined at the design level.

Experimental evaluation

The core test architecture described above is currently being evaluated on a sample chip including a 16 K × 16 bits SRAM embedded memory manufactured by STMicroelectronics using a mixed/power 0.18 µm library and on a M50FW040 Flash embedded memory produced by STMicroelectronics, which size is 4 Mbit and it is divided into 8 blocks with a 8 bit word parallelism. The M50FW040 device presents some particularities that imply a specific Flash Manager design:

- specific codes are needed to access the memory;
- the addressing phase is divided into two distinct steps due to the presence of only 10 address bits: the address is provided by sending first the 10 less significant bits, then the 9 remaining bits.

In both cases, the test program resides in a RAM memory, loaded from the outside through the IEEE 1500 ports.

Considering the SRAM oriented architecture, the size of the test program for the 12N adopted March algorithm is 43 4-bit words.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

Tab. III: A portion of the Test Program.

<table>
<thead>
<tr>
<th>March symbol</th>
<th>Instruction</th>
<th>Address</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>RST_ADD</td>
<td>N</td>
<td>0110</td>
</tr>
<tr>
<td>{}</td>
<td>INV_DBG</td>
<td>N+1</td>
<td>1000</td>
</tr>
<tr>
<td>R1</td>
<td>READ</td>
<td>N+2</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td>INV_DBG</td>
<td>N+3</td>
<td>1000</td>
</tr>
<tr>
<td>W0</td>
<td>WRITE</td>
<td>N+4</td>
<td>1010</td>
</tr>
<tr>
<td>}</td>
<td>BNE -5</td>
<td>N+5</td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N+6</td>
<td>0101</td>
</tr>
<tr>
<td></td>
<td>SETTIME</td>
<td>N+7</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td>255</td>
<td>N+8</td>
<td>1111</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N+9</td>
<td>1111</td>
</tr>
<tr>
<td></td>
<td>PAUSE</td>
<td>N+10</td>
<td>1111</td>
</tr>
<tr>
<td>↓</td>
<td>SET_ADD</td>
<td>N+11</td>
<td>1110</td>
</tr>
<tr>
<td>{R0</td>
<td>READ</td>
<td>N+12</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td>INV_DBG</td>
<td>N+13</td>
<td>1000</td>
</tr>
<tr>
<td>W1</td>
<td>WRITE</td>
<td>N+14</td>
<td>1010</td>
</tr>
<tr>
<td></td>
<td>INV_DBG</td>
<td>N+15</td>
<td>1000</td>
</tr>
<tr>
<td>}</td>
<td>BNE -5</td>
<td>N+16</td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N+17</td>
<td>0101</td>
</tr>
</tbody>
</table>

The core implementing the proposed test processor architecture was modeled in VHDL for an amount of about 3,000 lines of code, then synthesized with Synopsys Design Compiler.

The total area occupied by the additional logic for test and diagnosis is reported in table IV. The Memory Adapter introduces the largest overhead due to the test registers it includes. The total area overhead introduced by the programmable BIST amounts to about 2.1% of the memory area. In table III the TAP Controller and the TAP have not been considered since they are not related to a single core, but shared among multiple cores present in the SOC. Anyway, their size amounts to about 800 gates.

It is interesting to note that the programmable BIST approach proposed in this paper requires a negligible extra area overhead with respect to the one introduced by the hardwired BIST approach [25][54] which requires 6,913 gates for the same memory type with the same technology.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

The proposed approach is also comparable in terms of area overhead with a different BIST and self diagnosis approach [14], where for a 128 Kb SRAM memory the area overhead amounts to 2.4% of the memory area.

The programmable BIST approach does not introduce any temporal overhead and it guarantees an at-speed test with a 40MHz clock frequency.

<table>
<thead>
<tr>
<th>Component</th>
<th>Programmable BIST [# of equivalent gates]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrapper</td>
<td>2,944</td>
</tr>
<tr>
<td>Control Unit</td>
<td>760</td>
</tr>
<tr>
<td>Memory Adapter</td>
<td>4,027</td>
</tr>
<tr>
<td>ROM</td>
<td>220</td>
</tr>
<tr>
<td>TOTAL</td>
<td>7,951</td>
</tr>
</tbody>
</table>

In the case of the Flash memory core, the size of the test program for the word-oriented March-FT algorithm [53] is 53 4-bit words, and for the basic March-FT [51] the test program length is 33 4-bit words.

Considering the word-oriented March FT algorithm, with respect to M50FW040 model, the overall area overhead stemming from the additional DfT logic is about 0.2% of the full memory area.

Table V shows also that the application of the basic March FT algorithm causes a marginal reduction of the introduced area overhead, due to the decrease of the test program length and to the use of a single databackground value. This result underlines the high flexibility of the proposed test architecture: a new test algorithm can be introduced by changing only the databackground description in the Memory Adapter unit and updating the test program stored in RAM.

In order to evaluate the test application time overhead with respect to an alternative hardwired approach, the test program has been executed and then the application of the same March algorithm directly simulated to the pins of a hypothetical stand-alone chip.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

The complete test program execution time requires 6.97 seconds for each block. This time is heavily conditioned by the specific access timing of the M50FW040 model, especially by the erase operation time, which requires about 0.75 sec per block.

The simulation of the application time to test a stand-alone chip equivalent to the same memory flash model requires 6.59 seconds. Comparing the times it can be stated that the time overhead is limited to about 6% of the complete test time.

<table>
<thead>
<tr>
<th>Component</th>
<th>Word-oriented March FT</th>
<th>March FT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[# of gates]</td>
<td>[# of gates]</td>
</tr>
<tr>
<td>TAP</td>
<td>786</td>
<td>786</td>
</tr>
<tr>
<td>TAP controller</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Wrapper</td>
<td>992</td>
<td>992</td>
</tr>
<tr>
<td>Control Unit</td>
<td>624</td>
<td>624</td>
</tr>
<tr>
<td>Memory Adapter</td>
<td>1,258</td>
<td>1,242</td>
</tr>
<tr>
<td>Flash Manager</td>
<td>307</td>
<td>307</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>4,246</strong></td>
<td><strong>4,155</strong></td>
</tr>
</tbody>
</table>

In a second set of experiments, considering the March FT algorithm, the internal time overhead grows to about 12% of the complete test time, as a consequence of the reduced number of erase operations executed during the test. Therefore, it has been experimentally proved that the time overhead does not limit the efficiency of the test, and that an at-speed execution can be supported by the proposed architecture.

3.1.2 Processor cores

A custom Infrastructure IP is described in this paragraph, designed to support the execution of Software-based Self-test procedures addressing the test and diagnosis of processor cores. In the SoC manufacturing test contest, the contribution of this document with respect to the approached topic is twofold: it provides the description of a programmable architecture oriented to diagnosis and it delineates the guidelines for including test access structures and define high level test application protocols. More in general, the proposed approach aims at the definition of an Infrastructure IP able to
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

provide full control on the upload and activation of software-based Self-test procedures, as well as on result compaction and retrieval.

The following constraints often exist while attacking the embedded processor core test:

- no modifications on the processor internal structure are allowed
- test must be performed at the same working frequency of the processor itself
- a low-cost ATE is in charge of performing the test, through a low-speed interface.

The overall structure of the proposed architecture is reported in figure 19. The I-IP circuitry is capable of taking the control of the processor in order to:

- upload the test-program in the code memory
- launch the Self-test procedure
- observe and compact the test results
- transfer the final test results to the ATE.

All these performed tasks are compatible with the defined constrains.
The Infrastructure IP internal structure

The overall test architecture, shown in figure 19, highlights how the wrapper interfaces the ATE to the Infrastructure IP. By receiving high-level commands, the I-IP circuitry is able to upload the test program into the processor code memory, control the Self-test execution and provide the final results when the test finishes. Accomplishing these objectives without introducing modifications in the internal processor core structure is a fundamental goal for many reasons:

- introducing changes into the processor core is often impossible since the internal structure is not available
- even when the internal knowledge of the processor core is available, its modification is a complex task requiring skilled designer efforts
- the modification of the internal structure could negatively affect the processor performance.

The set of criteria identified for a not invasive HW/SW technique are the following:

- test program uploaded by reusing the system bus
- activation of self-test procedures exploiting the interrupt features of the processor core
- signature generated by enhanced test programs
- results collected in a MISR module connected to the system bus, able to compress the signature sent during the test program execution.

The details of the I-IP circuitry devoted to efficiently access the processor core and the test program modifications are given taking into considerations the 1500 compatibility requirements. The internal structure of the I-IP and its connections to the processor core system are reported in figure 20.

**Test program upload:** To upload test program into the memory portion devoted to store the Self-Test code is the first task of the approach. In order to write the test
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

program code and data into the instruction and data memory without any particular requirements (e.g., dual port memories, DMA controllers, etc.), a strategy based on the reuse of the system bus is proposed. When the ATE sends an upload command, the signals to take the control of the bus are generated in two ways:

- a SELECT signal drives a selection circuitry directly connecting the I-IP to the memory
- the I-IP takes the control of the bus by directly acting on the processor functionalities by means of driving its address, data and control ports to high-impedance or running special procedures to move data from the I-IP to the memory.

In figure 20, the UPLOAD module of the I-IP is the module in charge of such signal generation.

Self-test procedure activation: assuming that the processor supports the interrupt mechanism, the basic idea is to transform the self-test program into an interrupt service procedure. In such a way, the address of the uploaded Self-test program is stored in the slot of the Interrupt Vector Table corresponding to the interrupt triggered as soon as the ATE sends the activation command. The interrupt signals are managed by the wrapper circuitry that converts the high-level command coming from the ATE into the activation sequence of the processor interrupt mechanism. The complexity of the wrapper module in charge of activating the self-test procedure depends on the interrupt mechanism supported by the processor: if the auto-vectorized interrupt protocol is available, the I-IP simply has to activate the proper interrupt signals. Otherwise, the I-IP must be able to activate an interrupt acknowledge cycle, and provide the processor with the proper interrupt type.
In figure 20, such architecture-dependent circuitry is the Test Activation module.

**Signature generation and result monitoring:** Signature generation and results monitoring involves the different aspects of the same problem: to obtain information about the test execution. Software-based self-test procedures include instructions targeted at activating possible faults and transferring their effects on registers or memory locations. However, other instructions need to be added to further transfer the fault effects to some easily accessible observability point (e.g., an external port, or a specific location in memory). In other words, the test program should include instructions writing test signatures to an observability point.

Therefore, the number of signatures sent to the observability point depends on the test code. As a great number of test data is normally produced, and the ATE can hardly observe them continuously, the adoption of strategies able to compact these data and send to the ATE the final signature, only, is mandatory. A possible solution consists in the use of a Multiple Input Shift Register (MISR) computing and storing the final test signature. MISR characteristics (e.g., length, primitive polynomial) must be tuned to ensure a sufficiently low percentage of aliasing and an acceptable silicon area overhead.
The \texttt{RESULT} module of the I-IP, shown in figure 20, includes the MISR and its control circuitry. Considering processor models adopting memory mapped I/O addressing, the \texttt{RESULT} module is connected to the system bus and seen like a memory location: signatures to be compressed are sent by using generic transfer instructions. Otherwise, the \texttt{RESULT} module is connected to a port of the processor core and accessed by specific custom instructions.

\textit{IEEE 1500 Wrapper description}

1500 compliant wrapper behavior is defined by the IEEE 1500 SECT. This standard mandates the use of an internal structure composed of at least three scan chains:

- \textit{Wrapper Instruction Register} (WIR). In test mode, it is used to enable other internal resources both to apply test data to and to read information from the core.

- \textit{Wrapper Bypass Register} (WBY). When selected by the WIR register, it can be used to bypass the signal coming from the Wrapper Serial Input (WSI) directly to the Wrapper Serial Output (WSO).

- \textit{Wrapper Boundary Register} (WBR). It is compliant with the IEEE 1149.1 standard and allows stimuli application. It is connected to each input/output pin and suitable to perform interconnection testing.

The use of additional scan chains is allowed by the standard. They are in charge of programming internal test structures, starting the test in a given mode, and reading results by sending back both instructions and data. Two additional Wrapper registers have been included, the \textit{Wrapper Control Data Register} (WCDR) and the \textit{Wrapper Data Register} (WDR).

WCDR is an instruction register: it sends commands to the I-IP that consequently controls the test flow. Such commands cover the following operations:

- to enable the test program to be stored into the instruction memory reading one instruction at time from the Data port (LOAD\_TEST\_PROGRAM)
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

- to start the Self-test by forcing the interrupt activation sequence (RUN_TEST)
- to provide information about the status of the test procedure (POLLING)
- to enable the transfer of test results to the Data port (READ_RESULTS).

WDR is a data register: when required by a command, it is in charge of supplying the I-IP with the data or to store results to be serialized. In particular, it is devoted to:

- store the test program instruction to be uploaded in the Instruction memory
- store the test status, allowing the ATE to poll for test end
- store the results of the Self-test coming from the MISR.

The overall wrapper structure is shown in figure 21.

![Diagram of the wrapper architecture](image)

By exploiting the above introduced wrapper commands, the test protocol summarized by the following pseudo-code can be adopted:

1. LOAD_TEST_PROGRAM
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

2. RUN_TEST

3. POLLING – monitor IIP state until the test reaches the end

4. READ_RESULTS.

Initially, the self-test code is loaded into an available memory module (1). As soon as the upload operation is concluded, the test program is started applying the proper activation sequence to the interrupt port (2). During the test execution, the test status is polled waiting for its end (3); finally, the ATE through the TAM can access results immediately after the self-test procedure reaches its end (4).

Experimental evaluation

Two case studies have been considered to evaluate the feasibility of the proposed solution and estimate its cost. The approach has been applied to two processor cores:

- Intel 8051 [55]
- SPARC Leon I [56].

Intel 8051: the Intel 8051 microcontroller uses two internal memories: a 64k-byte sized ROM memory and a 256-byte sized RAM memory for registers, stack and variables. On the contrary, programs are stored in the external RAM connected with the parallel ports. The adopted model includes the auto-vectorized interrupt handling circuitry. In this case, the test program is loaded into the external RAM memory exploiting a selection circuitry controlled by the SELECT signal from the UPLOAD module parallel port. The starting address of the test procedure, stored in the ROM memory in the portion reserved to the interrupt vector table, is accessed as soon as the TEST ACTIVATION module generates the interrupt sequence. Test signatures generated during the test execution are compressed by a 32-bit wide MISR included into the RESULT module that is connected to the parallel port P1 directly available in the microcontroller. The schematic view of the Intel 8051 case study is reported in figure 22.
**SPARC Leon I:** The considered SPARC Leon I model is provided with an internal 512Kb-sized PROM and the test program resides in the external RAM. It implements an auto-vectorized interrupt handling mechanism, similarly to the 8051, but in this case the memory access mechanism is memory mapped I/O.

For this processor model, a mechanism exploiting its memory organization has been used. As shown in figure 23, the I-IP is provided with a *code buffer* contained in the UPLOAD module. Such buffer can be seen by processor core as a set of memory locations in the I/Os space and used to store both instructions and data; the starting address of the test procedure, stored in the PROM memory together with the addresses of all the trap management subroutines, matches with the first location of this buffer. As soon as the **TEST ACTIVATION** module applies the interrupt signal sequence, the processor executes the instruction previously stored into the buffer with the following results:

- if the entire test program can be stored into the buffer, the test of the processor is completely stored in the buffer and directly executed from this location
• if the test program is too large to be stored into the buffer only,
  o a program suitable to upload the test program is stored, which execution
    permits to move a program part in the buffer to the external RAM (this
    process is repeated until the whole test program has been uploaded)
  o a program jumping to the starting address of the uploaded test program is
    executed to perform the test.

The test flow individuated is the following:
1. LOAD_BUFFER
2. RUN_TEST
   a. if (program_size > buffer_size)
      repeat from 1 until code_upload != complete
3. POLLING - repeat 3 until end_test=0
4. READ_RESULTS

Fig. 23: SPARC Leon I approach.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

To compress the test results generated during the program execution, a 32-bit wide MISR module is used. In this case the RESULT module is directly connected to the system bus and accessed like a memory location.

In order to evaluate the effectiveness of the approach in terms of hardware cost of the proposed architecture, the RT-level behavioral VHDL description of the I-IPs and their P500 wrapper have been synthesized using the Synopsys Design Compiler tool with a generic gate library. Results are shown in table VI. In the Intel 8051 case, the silicon area overhead is almost entirely due to the introduction of the 1500 wrapper, while the introduction of the I-IP to the support to the self-test approach (UPLOAD, TEST ACTIVATION and RESULT modules) results in less than 2% of the additional area. On the contrary, the weight of the I-IP in the SPARC Leon I case grows, due to the introduction of a 216 bit sized code buffer. Such a solution well suits for short test programs as the whole code is stored in the buffer and executed directly form the I-IP.

<table>
<thead>
<tr>
<th>Module</th>
<th># of equivalent gates</th>
<th>Intel 8051</th>
<th>SPARC Leon I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor core</td>
<td></td>
<td>25,292</td>
<td>65,956</td>
</tr>
<tr>
<td>I-IP</td>
<td></td>
<td>490</td>
<td>3,632</td>
</tr>
<tr>
<td>1500 Wrapper</td>
<td></td>
<td>1,580</td>
<td>2,263</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>27,362</td>
<td>71,841</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td></td>
<td>8.1 %</td>
<td>8.9 %</td>
</tr>
</tbody>
</table>

Using an in-house developed tool [25], a self-test code has been generated to validate the presented approach. In table VII, the characteristics of the used test are summarized in terms of length and fault coverage with respect to the stuck-at fault model. The results for the SPARC Leon I processor refers to a test procedure generated to test the processor pipeline.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

Tab. VII: test-program length and fault coverage.

<table>
<thead>
<tr>
<th>processor model</th>
<th>Program length</th>
<th>test time [CK]</th>
<th>SA FC [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original</td>
<td>modified</td>
<td></td>
</tr>
<tr>
<td>Intel 8051</td>
<td>846</td>
<td>1,234</td>
<td>8,768</td>
</tr>
<tr>
<td>SPARC Leon I</td>
<td>1,194</td>
<td>1,678</td>
<td>21,552</td>
</tr>
</tbody>
</table>

3.1.3 User-Defined Logic cores

The BIST structure proposed in this paragraph particularly suits for modular logic cores. Such a BIST circuitry has been designed to achieve two goals: on one hand we want to simplify the introduction of any modification in the pattern generation algorithm by exploiting programmable structures; on the other hand we aim at showing a viable solution for easy integration of logic BIST structures in the SoC test plan. For the second reason, the adopted architecture exploits the same IEEE 1500 compliant test standard interface just owned by the memory and processor test structures aforementioned in order to simplify the designer effort.

The Infrastructure IP engine

The BIST engine internal architecture is divided into the following functional blocks:

- a Control Unit to manage the test execution;
- a Pattern Generator to produce and apply the test patterns;
- a Result Collector to manage the memory access timing.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

In figure 24 a generic environment for the considered approach is presented: the A, B and C modules are parts of the same logic core and communicate among them in order to process inputs and generate outputs. In the picture, inputs and outputs of each module are considered separately to ease the test approach comprehension.

The Control Unit manages the test execution; by receiving and decoding commands from the control signals, this module is able to manage the test execution and the upload of the results. In particular, it covers three tasks:

- it receives from the data signals the number of patterns to be applied
- it drives the test_enable signal that starts and stops the test execution and provides the information about the end of the test
- it selects the result to be uploaded.

This choice allows easy reuse in different applications, like the application of test vectors generated according to different algorithms, or the test of logic cores with different characteristics.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

The Pattern Generator is in charge of the application of the patterns to the DUT and is composed of:

- an ALFSR module [33][34]
- a set of Constraints Generators (CG).

The ALFSR module generates pseudo-random patterns according to the chosen polynomial characteristics; for cores composed of many functional blocks, only one ALFRS circuitry can be employed. On the contrary, a Constraint Generator is a custom circuitry able to drive constrained inputs. The adoption of such blocks provides great improvements in terms of effectiveness of the applied test, where a particular state machine controls the behavior of the circuit. In the design of the pattern generation circuitry, four architectural situations can be identified:

a. the block under test does not have constrained inputs and the ALFRS size fits the input port width
b. the block under test does not have constrained inputs and the input port width is larger than the ALFRS dimension
c. the block under test does have constrained inputs and the ALFRS size fits the input port width
d. the block under test have constrained inputs and the input port width is bigger than the ALFRS dimension.

While in a) the designer task consists just in connecting the ALFRS output with DUT input port, in b), c) and d) scenarios more care in choosing connections is required. Respectively, designers have to

- replicate the ALFSR outputs to reach the input port width
- identify the constrained inputs to build the CG and connect the ALFRS output to the remaining inputs.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

- identify the constrained inputs to build the CG and replicate the ALFRS outputs to drive all the remaining inputs.

These three situations are shown in figure 24 where respectively \( w_B > w_{ALFSR} \), \( w_A < w_{ALFSR} + w_{CG_A} \) and \( w_C > w_{ALFSR} + w_{CG_C} \).

The Result Collector is in charge of storing and making the results reachable from the outside. It is composed of

- a set of MISR modules
- an Output Selector module.

The ability of MISR modules to compact information with a low percentage of aliasing makes them suitable to store the results of the test. In this approach each module under test is coupled with a MISR: whereas the size of the MISR cannot exceed a predefined size, a xor cascade has been used. Each MISR module is reachable from the outside by programming the Output Selector.

This organization allows reducing the re-design operations and supports the reuse of the internal structures: changing ALFSRs and MISRs dimension is a trivial task. Only the individuation of input constraints and the design of the Constraints Generator require a bigger effort to designers.

Fault coverage and diagnosis ability evaluation

To obtain high fault coverage and guarantee high ability in terms of fault location, three steps are needed:

1. Statement coverage and toggle activity evaluation
2. Fault coverage measure
3. Equivalent fault classes computation.

In the first step, pseudo-random patterns are applied to the RTL description of the modules composing the logic core and the measure of the percent number of VHDL lines executed is performed. Such measure, usually called statement coverage, together with
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

the calculation of the percent number of variables toggled by the patterns, called toggle activity, gives to the designer a first degree of confidence about the effectiveness of the generated patterns [57] and can be performed using a simulation tool. Until this step, the evaluated patterns can be generated using the VHDL description of the Pattern Generator or simply calculated with ad-hoc tools generating pseudo-random sequences. Figure 25 represents the first step.

![Diagram]

**Fig. 25:** Statement coverage and toggle activity evaluation loop.

The second step refers to the synthesized component and it can be performed using a fault simulator. In order to obtain reliable results, the design to be evaluated in this step should already include the Pattern Generator and the MISRs embedded into the Result Collector, as the final layout optimization will merge their circuitry with that of the device under test. Whereas the fault coverage reached is less than the required, three actions can be performed:

- apply a larger number of patterns
- modify the ALFSR or MISRs structure
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

- redefine the Constraint Generator where included.

While the first action does not require any modification in the designed circuitry, the number of patterns can be increased only until the time for test requirements are not exceeded. In this case, the flow backs to the first step with the evaluation of new patterns. This loop, reported in figure 26, ends when the desired fault coverage is reached or when exceeding the manufacturing constrains.

![Diagram](image)

**Fig. 26:** Statement coverage and toggle activity evaluation loop.

The thirst step aims at reaching high diagnosis ability: this characteristic of the performed test can be evaluated by means of the size of the equivalent fault classes [58]. Such measure allows establishing the precision in terms of fault location provided by the analyzed patterns. The size of the equivalent fault classes mainly depends on the ability
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

of the chosen patterns to produce a different syndrome for every fault possibly affecting the DUT.

To reach this purpose, a tool able to apply patterns and store the circuit response is needed. The collected information, by means of the obtained syndromes, can be used to build the so-called diagnostic matrix [59], allowing to identify the faults belonging to the same equivalent fault class.

To improve the diagnostic properties of the generated patterns, it is possible to operate in two ways:

- adding test patterns
- changing the test structure characteristics.

The 1500 wrapper module

The wrapper, shown in figure 27, contains the circuitry necessary to interface the test processor with the outside in a 1500 compliant fashion, supporting the commands for running the BIST operation and accessing to its results. The wrapper is compliant with the suggestions of the 1500 standardization group [4]. The wrapper can be connected to the outside via a standard TAP.

In addition to the mandatory components, the introduction of the following Wrapper Data registers is proposed:

- Wrapper Control Data Register (WCDR): through this register the TAP controller sends the commands to the core (e.g., core reset, core test start, the Status register read, etc.).

- Wrapper Data Register (WDR): it is an output register. The TAP Controller can read the test information stored into the status register.
Experimental evaluation

The outlined approach has been applied to a Reconfigurable Serial Low-Density Parity-Checker decoder core [60][61][62]. This core was developed by my institution in the frame of a project involving several semiconductor, equipment, and telecom companies; to make it more easily usable by core integrators, a test solution was required.

Low-Density Parity-Check (LDPC) codes are powerful and computationally intensive error correction codes, originally proposed by Gallagher [60] and recently rediscovered [61] for a number of applications, including Digital Video Broadcasting (DVB) and magnetic recording. LDPC codes can be represented as a bipartite graph shown in figure 28, where two classes of processing elements iteratively exchange information according to the Message Passing [62] algorithm: Bit Nodes (BN) correspond to the codeword symbols, while Check Nodes (CN) are associated to the constraints the code poses on the bit nodes in order for them to form a valid codeword; at each iteration, the reliability of the decoding decisions that can be made on basis of the exchanged information is progressively refined.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

![Bipartite graph for an LDPC code.](image)

Fully and partially parallel solutions for the implementation of the decoder exploit the regularity of the bipartite graph mapping directly CN’s and BN’s into hardware blocks; either graph edges are mapped to proper interconnect infrastructures or they are implemented by means of memory banks. The complexity of check and bit nodes is strongly related to the number of incoming/outgoing edges; additional complexity comes from the requirement of supporting different edge numbers that are typical in the most powerful, irregular codes.

In [61], the implementation of an LDPC decoder is proposed, based on the use of a shared memory that emulates the interconnection between BIT_NODEs and CHECK_NODEs. In [60], a further implementation is proposed, introducing programmability in the architecture proposed in [62]. In this approach, a configurable BIT_NODE and a configurable CHECK_NODE are described. Their ability consists in emulating more than one module, by mapping more “virtual” nodes to the two physically available processing elements; the interconnections are simulated by means of two “interleaving memories” and thanks to its reconfigurable characteristics, this decoder is able to support codes of different sizes and rates, up to a maximum of 512 check nodes and 1,024 bit nodes. A CONTROL UNIT is introduced in order to manage the memory access and the reconfiguration information. The schematic of this enhanced circuitry is reported in Figure 29. Additionally to the use of the showed core in conjunction with
external memories to achieve a serial reconfigurable decoder, it can also be adopted as the basic building block for the implementation of a fully parallel architecture.

![Architecture of the Reconfigurable Serial Low Density Parity Checker decoder](image)

**Fig. 29:** Architecture of the Reconfigurable Serial Low Density Parity Checker decoder [15]. The BIT_NODE (BN), the CHECK_NODE (CN) and the CONTROL_UNIT (CU) are connected to the two interleaved memories to perform error detection and correction during transmission of high data volumes.

As far as this design is considered, the analyzed logic core is partitioned in BIT_NODE, CHECK_NODE and CONTROL_UNIT modules. The characteristics of each module in terms of input and output port size is reported in table VIII. The test of the two interleaved memories and the buffers is not considered in this paper.

<table>
<thead>
<tr>
<th>Component</th>
<th>Input port size [bits]</th>
<th>Output port size [bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT_NODE</td>
<td>54</td>
<td>55</td>
</tr>
<tr>
<td>CHECK_NODE</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>CONTROL_UNIT</td>
<td>45</td>
<td>44</td>
</tr>
</tbody>
</table>

**Tab. VIII:** Input and output port size in bits.

The BIST engine has the following structure. The Control Unit contains a counter register (pattern_counter) on 12 bits, allowing to apply up to 4,096 patterns for each test.
execution, and generates a 2 bits signal connected with the Result Collector, in charge of selecting the output to be read.

The Pattern Generator is equipped with a 20-bit ALFRS module and only one Constraint Generator is connected both to the BIT_NODE and to the CHECK_NODE of the serial LDPC while the CONTROL UNIT does not need it. The Constraints Generator manages a 4 bits sized port that internally selects the data path into the circuitry: it allows applying a limited number of patterns when a small data path is selected, while holding selection values that maximize the used circuitry.

The Result Collector is composed of three 16 bit sized MISR modules, each one connected to the DUT outputs through a xor cascade, and a Output Selector, whose behavior is driven by the Control Unit.

The total area occupied by the DfT additional logic is reported in table IX, and has been worked out by using a commercial tool (Synopsys Design Analizer) using an industrial 0.13 µm technological library.

The TAM logic (which includes the Wrapper module) represents a fixed cost necessary to manage the chip-level test. Its area overhead can be quantified as the 16% of the global cost of the additional core-level test logic.

<table>
<thead>
<tr>
<th>Component</th>
<th>Area [µm²]</th>
<th>Overhead [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial LDPC</td>
<td>165,817.88</td>
<td>-</td>
</tr>
<tr>
<td>BIST engine</td>
<td>22,481.63</td>
<td>13.5</td>
</tr>
<tr>
<td>1500 Wrapper</td>
<td>4,566.94</td>
<td>2.8</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>192,866.51</td>
<td>16.4</td>
</tr>
</tbody>
</table>

The fault coverage percentage reached by this approach is reported in table X and refers to both Stuck At Faults (SAF) and Transition Delay Faults (TDF). Such results have been obtained employing a commercial fault injection tool (Synopsys Tetramax). In order to provide the reader with reference figures, the data related to the cases in which sequential and full scan patterns produced by a commercial ATPG tool (again Synopsys
Tetramax) are used. It is important to note that these patterns could not be easily applied to the core, if embedded in a SoC, while the BIST approach is very suitable to deal with this situation. The number of scan cells inserted is 75 for the BIT_NODE, 803 for the CHECK_NODE and 42, divided in two scan chains including 14 and 28 cells, for the CONTROL_UNIT. These values have been calculated using a SUN workstation equipped with a SPARC V8 microprocessor and the CPU times reported in the above table working at 431.03 MHz in the case of the BIST engine approach (at-speed testing) and at 100 MHz (supposed ATE frequency) in the Sequential and Full scan approach.

With respect to the Sequential and Full Scan approaches, the use of the BIST approach is desirable for at least the following reasons:

- the fault coverage reached is higher than Sequential patterns coverage and comparable with Full Scan
- the BIST patterns are the same for all modules to be tested, so that they can be tested simultaneously
- the test time is significantly lower for the BIST approach than for the full-scan one
- such patterns are generated and applied one for each clock cycle and results read in the end of the execution, while Sequential and Full scan patterns have to be sent serially by the ATE and results uploaded serially after each operation, thus drastically increasing the ATE storage requirements
- the test patterns are applied by the BIST engine at the nominal frequency of the circuit while the Sequential and Full scan patterns are applied at the ATE frequency that could be lower, guaranteeing more efficiency in the fault coverage.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

Tab. X: Fault coverage figures.

<table>
<thead>
<tr>
<th>Component</th>
<th>BIST patterns</th>
<th>Sequential patterns</th>
<th>Full scan patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAF</td>
<td>TDF</td>
<td>SAF</td>
<td>TDF</td>
</tr>
<tr>
<td>BIT NODE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faults [#]</td>
<td>7,532</td>
<td>7,532</td>
<td>7,836</td>
</tr>
<tr>
<td>FC [%]</td>
<td>97.8</td>
<td>93.8</td>
<td>98.5</td>
</tr>
<tr>
<td>clock cycles</td>
<td>4,096</td>
<td>11,340</td>
<td>21,248</td>
</tr>
<tr>
<td>CPU time</td>
<td>-</td>
<td>489 sec</td>
<td>197 sec</td>
</tr>
<tr>
<td>CHECK NODE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faults [#]</td>
<td>86,104</td>
<td>86,104</td>
<td>89,412</td>
</tr>
<tr>
<td>FC [%]</td>
<td>91.6</td>
<td>82.9</td>
<td>93.1</td>
</tr>
<tr>
<td>clock cycles</td>
<td>4,096</td>
<td>8374</td>
<td>380,064</td>
</tr>
<tr>
<td>CPU Time</td>
<td>-</td>
<td>~ 54 h</td>
<td>428 sec</td>
</tr>
<tr>
<td>CONTROL UNIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faults [#]</td>
<td>3,038</td>
<td>3,038</td>
<td>3,216</td>
</tr>
<tr>
<td>FC [%]</td>
<td>97.5</td>
<td>89.8</td>
<td>98.6</td>
</tr>
<tr>
<td>clock cycles</td>
<td>4,096</td>
<td>4,860</td>
<td>16,965</td>
</tr>
<tr>
<td>CPU time</td>
<td>-</td>
<td>2422 sec</td>
<td>91 sec</td>
</tr>
</tbody>
</table>

In table XI, the measure of the performance reduction in terms of frequency lost is reported. This is due to the introduction of the BIST engine and the wrapper. This value is compared with those coming from the analysis of the Sequential and Full Scan approach, supposing that:

- for the Sequential approach, patterns are applied using a standard 1500 wrapper
- for the Sequential approach, patterns are applied using a standard 1500 wrapper and introducing into the design multiplexed scan cells.

Tab. XI: Performance reduction for the investigated approaches.

<table>
<thead>
<tr>
<th>Frequency [MHz]</th>
<th>Original design</th>
<th>BIST engine</th>
<th>Sequential approach</th>
<th>Full scan approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>438.6</td>
<td>431.03</td>
<td>434.14</td>
<td>426.62</td>
<td></td>
</tr>
</tbody>
</table>

Finally, table XII shows the size of the equivalent fault classes for the three components obtained for the BIST engine, Sequential patterns and Full Scan approach
applying the number of patterns reported in table X. That result has been obtained exploiting an in-home developed tool in C language described in Appendix A.

Tab. XII: Equivalent fault classes maximum and medium size obtained by the investigated approach.

<table>
<thead>
<tr>
<th>Component</th>
<th>BIST patterns Max size</th>
<th>BIST patterns Med size</th>
<th>Sequential patterns Max size</th>
<th>Sequential patterns Med size</th>
<th>Full scan Patterns Max size</th>
<th>Full scan Patterns Med size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT_NODE</td>
<td>3</td>
<td>1.2</td>
<td>7</td>
<td>4.4</td>
<td>3</td>
<td>1.6</td>
</tr>
<tr>
<td>CHECK_NODE</td>
<td>4</td>
<td>1.9</td>
<td>12</td>
<td>6.9</td>
<td>7</td>
<td>2.7</td>
</tr>
<tr>
<td>CONTROL_UNIT</td>
<td>2</td>
<td>1.3</td>
<td>8</td>
<td>5.1</td>
<td>2</td>
<td>1.3</td>
</tr>
</tbody>
</table>

3.2 System layer

The contribution given by this document regarding the system layer is extensively reported in this paragraph, describing in details a centralized embedded diagnostic manager, implemented as an Infrastructure IP (I-IP) able to control a diagnosis-oriented Test Access Mechanism (TAM) used to interconnect the I-IP included in the SoC design to apply a self-test procedure introduced at the core layer.

This architectural solution uses IEEE 1500 compliant structures and suits for medium/large sized memories: the I-IP, controlled through a TAP port, sends commands to each BISTed core included in the design and collects information about the whole set of failures by executing a diagnostic program fetched from a code memory. The main advantages are the following:

- the latency introduced by the ATE in supporting the diagnosis flow is practically reduced to zero, as all retrieved diagnostic information are internally elaborated and then communicated outside the SoC
- besides the transparency in managing diagnostic procedures, the proposed solution affords diagnostic time reduction, working at an ATE-independent frequency
• a low cost ATE is required, as the proposed approach asks for limited pattern storage and computation.

3.2.1 Diagnosis requirements at the system layer

A generic diagnostic protocol consists in the execution of several parametric phases, depending on the number and type of faults affecting the device. For this reason, the adoption of an I-IP able to manage autonomously the diagnostic procedures for SoCs including several memories is proposed.

Typical BIST modules equipped with diagnosis facilities (also referred as diagnostic BIST or dBIST modules) implement the flow reported in figure 18 in pseudo-C code. At first, the dBIST hardware is programmed for a complete scan of the memory; the variable error_detected is introduced for discriminating a fault-free condition, and it is initialized at True for allowing the first complete test run, at least. The actual test consists in the execution of March steps. When a fault is detected, diagnostic data is downloaded (Download_diagnostic_inf procedure) and the dBIST is reprogrammed to run the test until the step corresponding to the last detected fault.

In SoCs, diagnostic procedures mandatorily require to be executed within physical constraints forced by technologies (i.e., power dissipation bounds), as it commonly happens during the test scheduling definition. However, with respect to test scheduling approaches, decisions have to be dynamically taken during diagnosis, as fault locations cannot be forecasted a priori. In the described approach, such decisions, usually taken by the ATE, are directly taken by the chip according to the user defined algorithm.

In this context, the use of standard interfaces to access BIST modules is highly desirable for two aspects. First, the use of homogeneous structures strongly facilitates test interoperability of embedded cores; moreover, the information about test are moved to an higher level of controllability. 1500 Wrappers offer the possibility to access each core independently through the definition of test primitives, that finally can be combined in system level descriptions for SoCs’ test strategies. Two basic test primitives for 1500 access are introduced:
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

- **SEND** <data, port>, for the application of data to the indicated port
- **READ** <result, port>, for fetching diagnostic data from the port to the outside.

In order to reduce the costs of memory diagnosis in SoCs, a suitable environment composed of an Infrastructure IP and a diagnosis-oriented TAM structure is proposed.

Considering systems where each memory core, or cluster of memory cores, is equipped with a dBIST, the I-IP is capable of controlling autonomously the diagnostic procedures. It executes a program coded in a dedicated instruction set and stored in a code buffer. This program includes test and diagnosis sections: the test segment, based on a chosen scheduling strategy, is initially executed. After that, the set of detected faulty cores is analyzed and a subset of dBISTs in the system is re-run according to predefined physical constraints and core priorities. The code devoted to diagnosis describes the reprogramming procedures of the selected cores and programs the I-IP to observe periodically the status of the system: as soon as a dBIST ends the current programmed step, pending dBIST requests are processed in order to individuate the better choice.

Both to ease the internal organization of the I-IP and to reduce the latency in the diagnostic procedures, a suitable diagnosis-oriented TAM architecture is used. Figure 30 shows a conceptual view of the approach.
3.2.2 The diagnosis-oriented TAM

The adopted TAM architecture can be theoretically divided in two layers.

The first one is the Core Layer: a suitable wrapper design is proposed, especially designed to speed-up the communication of diagnostic information outside the memory core. This wrapper structure, shown in figure 31, is fully compliant with the IEEE 1500 SECT and efficiently supports the typical phases of a diagnostic process:

- dBIST initialization
- algorithm execution
- polling of the test status
- failure information extraction.
In addition to the 1500 mandatory components, the wrapper architecture includes two registers that can be serially loaded and read through the \textit{tsi} and \textit{tso} signals, respectively. These registers are connected in parallel to the diagnosis ports of the dBISTed memory core: \textit{Wrapper Command Data Register (WCDR)} to send instructions and \textit{Wrapper Data Register (WDR)} to read or write diagnostic data.

The second layer, named \textit{System Layer}, guarantees high flexibility in the execution of the diagnosis phases for complex SoCs: the defined TAM, that exploits the reusability of control signals of the wrapper structure, gives to the test equipments the ability to manage the diagnosis of the whole system taking into consideration the diagnosis requirements summarized in the previous section. It is based on two busses:

- \textit{Control bus}: it reaches each core included in the system and carries the information to manage the scan chain loading

- \textit{Data bus}: it transports the data to or from the cores.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

The set of memory cores to be diagnosed is distributed into many subgroups, organized as follows:

- each core belonging to a subgroup is connected to a shared $WIP$
- each core included in a subgroup receives data from a common $tdi$ signal
- each core in each subgroup writes data over an independent $tdo$ signal, shared with the cores placed at the same position in the other subgroups.

As shown in figure 32 the number of required TAM wires for this solution is dependent on the number of subgroups and on the maximum number of cores included into a subgroup.

Even if their contribution can be limited by optimized floor planning strategies, additional TAM wires do not impact on the pin count. In fact, they are completely controlled by the Infrastructure-IP.

Fig. 32: schematic view of the System Layer
3.2.3 The I-IP structure

An Infrastructure-IP, called Diagnostic Manager, controls test and diagnosis procedures to be applied to each core included in the system. Such additional hardware is capable of driving efficiently the diagnosis-oriented TAM and performs the following tasks:

- it sends to a single core (or directly to a subgroup) commands and data needed to control the algorithm execution during every stage of the diagnostic flow
- it reads and stores the results retrieved in the end of each single diagnostic step execution
- it internally elaborates at run-time the parameters of the current diagnostic step, taking into consideration physical constraints individuated by designer.

The ATE controls the Diagnostic Manager through an IEEE 1149.1 TAP controller, which allows data stored in the internal structure to be transmitted out to the ATE. The internal architecture of the Diagnostic Manager is shown in figure 33.

Fig. 33: schematic view of the proposed I-IP

The 1500FSM module is a finite state machine (figure 34a) in charge of generating the signals of the Control bus and Data bus.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

At the circuit reset, the 1500FSM holds IDLE. As soon as the ATE sends the START command to enable the Diagnostic Manager, it activates the dBIST functionalities. In particular, at any given time, and for each controlled dBIST module, the 1500FSM can be in one of the following macro-states:

- **UPDATE**: a sequence of signals to be sent to a specified dBIST module port is generated
- **CAPTURE**: a sequence of signals for reading a specified dBIST module port is generated
- **IDLE/RUN TEST**: the 1500FSM waits for a known period while executing no operations.

The behavior of the 1500FSM is programmable: a defined test and diagnosis strategy stored in the Code buffer module resorting to a specific instruction set. The op-codes and instructions are shown in figure 34b.

Every instruction is identified by the first bit (NS, next state field), which determines the macro-state to reach. When a WAIT instruction is executed, the system rests for the time specified in the DELAY field; if a SEND or a READ instruction is executed, the
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

The macro-state reached is UPDATE or CAPTURE, respectively. For both these instructions the next fields correspond to the subgroup identifier (SID), the core identifier (CID) and the wrapper register identifier (WID). These instructions are applied to an entire subgroup if the broadcast field (BC) is asserted. The last field differentiates SEND and READ instructions as it represents a data to be sent (DATA) or the number of result bits to be downloaded (RESULT), respectively.

Fig. 35: the micro-states of the 1500FSM are shown in a). In b), c) and d) are shown the state sequences for wait, update and capture operation, respectively.

Multiple instances of the finite state machine have to be included in the Diagnostic Manager if the access protocol is not the same for every dBIST module.

The Code buffer module stores the instructions for the 1500 FSM. Its content is divided into two blocks: Test scheduling program and Diagnostic procedures.

The first block is executed as soon as the ATE enables the I-IP to work. This code portion is generated a-priori exploiting a test-scheduling algorithm.

When the test phase ends, two Diagnostic procedures are employed to perform the diagnosis of the system: the Subgroup polling procedure executes an interleaved access to subgroups, requesting the dBIST status. When one or more dBIST modules communicate the end of a diagnostic step, the Next step execution procedure is called. This procedure relies on the Response evaluator module, which provides the dBIST modules with the updated diagnostic parameters.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

The Subgroup polling procedure is restarted immediately after the execution of this code portion.

The selector module routes the generated control signals to the proper I-IP channel according to the SID field in SEND and READ instructions.

The Response evaluator module is in charge of elaborating the information extracted from the dBIST modules and selects the next diagnostic step: in practical terms, it activates the Next step execution procedure and supports the P5000FSM module in the choice of the new parameter (SID, CID, WID and DATA). It includes three registers involved in managing priorities and collisions in real-time (priority register, execution register and pending register) and stores information about physical constraints and conflicts.

The Result buffer module collects all the diagnostic signatures extracted during the diagnostic flow execution. Immediately after writing a result, the status of the Diagnostic Manager is updated and the presence of failure data is signaled to the ATE. The download of these results (fault locations and failed memory words) is performed independently from the execution of any other test of the system. In the meantime, the I-IP elaborates the collected data to prepare the next diagnosis step.

3.2.4 Experimental evaluation

In order to prove the effectiveness of the proposed approach, a case study has been analyzed. The issue of this section is mainly to demonstrate the time reduction capability of the proposed framework. However, the introduced advantages are not limited to this aspect, as this approach allows for reduced ATE storage, frequency and easier diagnostic procedure without external data computation.

A generic SoC including 20 memory cores of different size has been considered as a case of study (see table XIII). Each core is equipped with the dBIST architecture detailed in [17].
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

**Tab. XIII:** the target SoC.

<table>
<thead>
<tr>
<th>Core</th>
<th>Addr (bit)</th>
<th>Data (bit)</th>
<th>Test length (ck cycles)</th>
<th>Test power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, …, 7</td>
<td>16</td>
<td>32</td>
<td>720,896</td>
<td>320</td>
</tr>
<tr>
<td>8, 9, …, 14</td>
<td>14</td>
<td>16</td>
<td>180,224</td>
<td>280</td>
</tr>
<tr>
<td>15, 16, …, 19</td>
<td>12</td>
<td>16</td>
<td>45,056</td>
<td>240</td>
</tr>
</tbody>
</table>

The I-IP and TAM were described in VHDL language resorting to about 3,000 code lines and adapted to the case study:

- the I-IP manages 5 groups, each one including 4 memory cores
- the instruction size is 16 bits and Code buffer is intended to be a ROM memory
- the Result buffer is a RAM memory (256 bytes).

The test program executed by the I-IP as the first step of the diagnostic procedures is obtained applying the scheduling algorithm proposed in [41] and it occupies 53 instructions. The Code buffer also stores the diagnosis management routines, resorting to 1 instruction for the Subgroup polling procedure and 3 instructions for the Next step execution procedure. Table XIV resumes the size of each module in the I-IP in terms of equivalent gates. With respect to the considered system, the introduced area overhead is less than 0.1%.

**Tab. XIV:** Diagnostic Manager area occupation

<table>
<thead>
<tr>
<th>I-IP module</th>
<th>Occupation [gate #]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500FSM</td>
<td>2,479</td>
</tr>
<tr>
<td>Code buffer</td>
<td>1,932</td>
</tr>
<tr>
<td>Result buffer</td>
<td>1,632</td>
</tr>
<tr>
<td>Response evaluator</td>
<td>2,740</td>
</tr>
<tr>
<td>Selector</td>
<td>158</td>
</tr>
<tr>
<td>Wrapper</td>
<td>994</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>9,935</strong></td>
</tr>
</tbody>
</table>

The estimation of the ability in time reduction for diagnosis was worked out considering 4 significant fault sets including typical memory defects (see table XV).
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

Tab. XV: defects localization in the memory cores for the considered examples

<table>
<thead>
<tr>
<th>Defect type</th>
<th>column</th>
<th>row</th>
<th>spot</th>
</tr>
</thead>
<tbody>
<tr>
<td>case 1</td>
<td>0, 1, 2, 3</td>
<td>4, 5</td>
<td>6, 10, 11</td>
</tr>
<tr>
<td>case 2</td>
<td>8, 9, 10</td>
<td>11, 12</td>
<td>0, 1, 4, 7, 18</td>
</tr>
<tr>
<td>case 3</td>
<td>-</td>
<td>15, 16, 17</td>
<td>1, 9, 16</td>
</tr>
<tr>
<td>case 4</td>
<td>-</td>
<td>-</td>
<td>0, 8, 15, 19</td>
</tr>
</tbody>
</table>

The diagnosis times for each scenario are calculated adopting the proposed approach, and compared with two classical ATE based solutions in table XVI:

a. serial based: every memory core is connected to a unique serial chain [38]; this solution requires 5 additional pins to manage the TAP controller as in the I-IP solution

b. subgroup based: 5 chains, each one including 4 dBISTed cores, are alternatively controlled exploiting the proposed TAM architecture; such solution requires 5 input pins and 4 output pins more than the first one.

Although the I-IP approach is based on the diagnosis-oriented TAM, the pin count is still 5 as in the approach B. The considered ATE frequency is 20 MHz, while the nominal frequency of the system is set to 500 MHz. A maximum power constraint of 720 mW is considered in the investigated SoC.

Tab. XVI: diagnosis execution time

<table>
<thead>
<tr>
<th>Faulty steps #</th>
<th>Test &amp; Diagnosis time (ms)</th>
<th>Time Gain %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>4,616</td>
<td>1,982.3</td>
</tr>
<tr>
<td>2</td>
<td>2,062</td>
<td>250.9</td>
</tr>
<tr>
<td>3</td>
<td>520</td>
<td>25.7</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>10.9</td>
</tr>
</tbody>
</table>

Starting from the obtained results, the following aspects have been underlined:
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

- the time for diagnosis is always reduced with respect to both ATE based solutions as
  - the use of the I-IP reduces the number of low-frequency ATE accesses to memory cores
  - the I-IP assumes internally the functionality of the TAP during the diagnostic session, thus reducing the number of clock cycles needed
- the benefit of the proposed approach depends on the faults topology and higher gains are obtained when dealing with a large number of faults in small memory cuts (scenario 2 and 3), as in that cases the diagnosis time is affected more by reprogramming operations than by the length of re-executions.

3.3 Application level

The architecture and characteristics of a software platform for increasing the design-to-test flow automation are presented. Support tools helping during the different phases of the design and test of a SoC must allow to

- early evaluate the advantages and costs of different test solutions
- support the exchange of test information between design and test
- validate the selected test architecture and strategy
- automate the generation of the final test program to be loaded on the ATE.

The tool presented in this paper, named SoC Test Automation Tool or STAT, covers these requirements. As graphically shown in figure 37, it permits basically:

- the quick validation and evaluation of different test solutions in terms of required test application time, providing test engineers with the possibility to explore the efficiency of a set of test architectures for a SoC in early phases of the design flow.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

- the automatic generation of platform-independent STIL test program to be fed into the Automated Test Equipment.

A prototypical version of STAT focusing on IEEE 1500 compatible cores has been realized; the functionalities and feasibility of the methodology have been proven on some benchmark SoCs.

STAT is a multilayer platform that allows introducing the description of the relevant test characteristics of:

- cores, in terms of DfT structures, protocols and test patterns;
- TAMs, in terms of type of chip test interconnections, bus width, and test scheduling;
- ATEs, in terms of equipment availability.

Given these descriptions, it is able to:

- perform a check on the compatibility between the described test architecture and the existing constraints, e.g. in terms of available ATE features or power consumption during test;
- quickly evaluate the SoC test time, when a valid test configuration is described;

Fig.36: the STAT platform as evaluator of test architectures and strategies, and STIL program generator
automatically generate the SoC test program in a defined language, including the waveforms to be applied and expected to be read by the ATE for the whole test execution.

STAT is designed to be a flexible environment that can read multiple libraries and templates, allowing the user to describe a range of different test configurations, easily switching from one to another and helping reusing parts of existing test projects in new configurations.

Figure 37 shows a conceptual view of the STAT platform. The rectangles in the upper side of the scheme represent the description of the circuit under test and of the ATE constraints, following the three-layer subdivision. The Test schedule contains information about the synchronization and time sequence of the core tests, while User constraints define the procedural rules for the overall system test. The bottom part of the scheme reports the STAT outputs.

These aspects will be described in detail in the following paragraphs.

3.3.1 Test structures and strategy description

At the core layer, STAT manages a description of the DfT architecture and test strategy for each core, which can be given in the STAT metalanguage or read from STIL.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

or CTL (IEEE 1450) format files. The functional cores can fall in the following categories:

- microprocessors / microcontrollers, that can be tested
  - structurally, resorting to scan chains and suitable test vectors, or
  - by introducing functional software-based self-test techniques [24], possibly exploiting suitable I-IPs [63]

- memories, usually tested by means of integrated BIST, which need to be controlled and polled from the outside according to a core-specific protocol

- sequential or combinational logic blocks, whose test resort to
  - application of patterns accessing to their primary I/O ports, or
  - logic BIST, or
  - scan chain insertion (for sequential cores, only).

The core test description managed by STAT includes:

- test length
- estimated power consumption
- BIST commands, parameters and functional procedures, if present
- test patterns, if needed.

The patterns may have been expressly generated manually or via an ATPG, or come along with third-parties supplied cores. It is possible to define additional patterns for implementing the interconnections test [64].

Each core should be equipped with an IEEE 1500 compatible wrapper in order to implement a common test interface; wrapper structure description is also included among the STAT inputs. When the chip includes unwrapped cores or distributed logic,
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

the IEEE 1500 structures of the other cores allow testing these parts resorting to interconnection test techniques.

The System layer deals with interconnections between the test structures (TAM) and the circuit pins. The core test structure can be interconnected through the simple IEEE 1500 standard serial line, but in many cases higher bandwidths are needed and buses for parallel test access are inserted. Among the different TAM structures known in the literature, at present STAT can manage the following ones

- **Multiplexing**, when the available test bus is completely assigned to one core at a time, in turn

- **Daisy chain**, when the test bus serially connects all the cores that can be set in bypass mode

- **Distribution**, when the test bus is partitioned between the cores [65]

- **Test Bus**, (a compromise between multiplexing and distribution architectures) where a partitioned TAM is connected to more than one core [40]

- **Test Rail**, a combination of daisy chain and distribution architectures [38].

A proper metalanguage has been defined to describe the TAM configuration, by selecting one of the currently implemented structures; different or hybrid TAM configurations are planned to be introduced with further developments of the tool.

At this level the user is given the possibility to introduce constraints defining test priorities or incompatibilities, and to specify the test scheduling, i.e., the sequence in time of the cores tests execution. The easier way to explicit the scheduling is to list the cores in their test beginning order.

At the ATE level, the STIL templates describing waveforms and external device signals are declared, including the logic values that must be kept on the functional device I/O ports not affected by the test procedure. Specific ATE technology-related constraints can be set within this layer (using a specific metalanguage file):
the constraints imposed on the circuit, such as
  - power limits
  - test conflicts

the constraints imposed by the available ATE, such as
  - maximum test program size (ATE per-pin memory)
  - multi-site test abilities.

### 3.3.2 Verification of a test plan

STAT helps verifying the effectiveness of a test plan, highlighting the criticalities that may rise at the test layers intersections: contradictions among imposed constraints in different layers can lead to unfeasible test solutions, which require changes in strategy and different user decisions.

More in details, the tool ensures that the circuit constraints (e.g., maximum test time, power consumption, etc.) are met, taking into account user-defined limitations in terms of priorities and mutual incompatibilities, and applying the specified scheduling.

This aspect becomes very useful when reutilizing existing strategies in new test environments, such as when introducing a new technology with different constraints (e.g., a lower power consumption limit can be incompatible with parallel execution of some tests) or when switching to a different ATE.

### 3.3.3 Evaluation of the test time

SoC test times are computed according to the specific scenario features, test scheduling and user constraints.

Each core-related test strategy involves the definition of a suitable protocol. The tool evaluates the actual testing time including test data management and transfer and TAP state machine control, as an estimation made only on the basis of the single core test duration would lead to inaccurate results and sub-optimal solutions.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

It is important to remember that in most cases data transfers to and from the ATE are not executed at the circuit nominal frequency, while BIST executions actually do. The evaluation capabilities of the tool can help test engineers in developing optimal test structures and strategies (BIST architecture, TAM insertion and partitioning, core grouping, scheduling, etc.).

3.3.4 Test program generation

Once the test architecture and strategy have been validated, STAT automatically produces the test program in STIL format.

The novelty of STAT consists in elaborating two test layers information also for the SoC test program description: by integrating data coming from very different descriptions (e.g., the test patterns for scanned cores, the control commands for BISTed, etc.) and applying the selected TAM protocol, it results in a comprehensive test program able to manage all phases of the test of the chip.

The generated programs are based on a set of STIL templates and incorporate the procedures that implement the protocols for test data transfer to and from the cores through the defined TAM. They contain the description of the actual waveforms that will be sent to and expected to be received from the I/O ports of the device under test.

In order to minimize the size of the STIL file, some pattern compression techniques have been introduced in the STIL generation flow. They rely on the two basic concepts of loops and macros. Loops compression is simply a pattern line with specific code to indicate the number of repetitions the tester must perform to apply a number of patterns to the DUT. This is often used to compress the idle cycles in the BIST patterns. Macro compression is a technique used to store a block of patterns that perform a particular function in separate ATE memory, which interleaves with the tester’s standard memory each time the macro is invoked. Figure 38 shows an example of two STIL macro definitions for IEEE 1500 wrapped cores, the first for programming the TAP state machine, and the second for serially loading data in one of the wrapper registers. They
both define a sequence of logical values that need to be consecutively applied to the TMS signal of the TAP.

![MacroDefs](image)

**Fig. 38:** example of STIL pattern compression through macro definitions and loops

### 3.3.5 Experimental evaluation

The current prototypical implementation of STAT manages IEEE 1500 wrapped cores belonging to the following categories:

- memories
- microprocessors and microcontrollers
- sequential/combinational logic cores.

The present version is implemented in about 1,400 lines of C code; STAT has been tested on some different SoC configurations. In figure 39 a case study SoC is presented, which is used to show how the tool can be fruitfully exploited by the test engineer. The introduced SoC is composed of:

- 2 open-source microcontroller cores A and B (mc8051 [66])
- 2 sequential logic cores (X, Y)
- 4 8x32K static RAM cores (1 – 4)
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

- 1 8x64K RAM core (5)
- 1 32x8K RAM core (6).

In order to evaluate and find a valid test configuration, no information is needed on the actual functional behavior and on the connections between the modules, or on the cores internal structure.

The test requirements for each core were defined at design time:

- Each of the microcontroller cores is sided by an Infrastructure IP [63] allowing the application of a functional test. The test program, generated adopting a deterministic technique [67] and executing 883 instructions, covers 89.47% of the stuck-at faults and takes up to about 5·10⁶ clock cycles; an optional complementary set of 80 scan test vectors (569 bits) was generated by an ATPG, raising the stuck-at fault coverage to 98%.

- Logic cores X and Y are equipped with a logic BIST structure applying 1024 and 864 pseudorandom patterns [68], respectively.
• Memory cores 1 – 4 are equipped with a hardwired BIST applying a 24N March algorithm [25][54]

• Memory cores 5 and 6 are equipped with programmable BIST circuits [17], initially programmed to apply an 8N and a 10N March algorithm, respectively.

The cores test structures are made accessible through IEEE 1500 compatible wrappers that connect each core test input/output signals to the TAM [35]. An IEEE 1149.1 5-pin TAP access port provides full access to the test structures; its description is included in the tool libraries.

To provide the reader with an example of input files for STAT, figure 40 presents the description of the test architecture and strategy (core layer) for logic core X, in an ad-hoc data format used by the prototypical STAT. The lines following the header WRAPPER INSTRUCTIONS contain information about the wrapper structure and the declarations of the BIST functions and commands. In particular, the width of each wrapper register is reported (wir, wcdr, wdr), together with the wir-level and wcdr-level instructions (possibly with the related wdr data exchange information).

The header STEPS introduces the test program steps definition for the selected core, using the instructions formerly declared.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

Fig. 40: description of the test architecture and strategy (core layer) for logic core X

<table>
<thead>
<tr>
<th>WRAPPER INSTRUCTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wir 3</code></td>
</tr>
<tr>
<td><code>000 bypass</code></td>
</tr>
<tr>
<td><code>001 WDR</code></td>
</tr>
<tr>
<td><code>010 WCDDR</code></td>
</tr>
<tr>
<td><code>wcdrr 3</code></td>
</tr>
<tr>
<td><code>000 syn_reset n</code></td>
</tr>
<tr>
<td><code>001 run_test n</code></td>
</tr>
<tr>
<td><code>010 upload_seed y in 16</code></td>
</tr>
<tr>
<td><code>011 upload_retro y in 16</code></td>
</tr>
<tr>
<td><code>100 download_retro y out 16</code></td>
</tr>
<tr>
<td><code>101 download_misr y out 16</code></td>
</tr>
<tr>
<td><code>110 upload_counter y in 16</code></td>
</tr>
<tr>
<td><code>wdr 16</code></td>
</tr>
<tr>
<td>STEPS</td>
</tr>
<tr>
<td><code>syn_reset</code></td>
</tr>
<tr>
<td><code>upload_seed 1010101010101010</code></td>
</tr>
<tr>
<td><code>upload_retro 1100110011001100</code></td>
</tr>
<tr>
<td><code>upload_counter 0000100000000000</code></td>
</tr>
<tr>
<td><code>run_test</code></td>
</tr>
<tr>
<td><code>idle 256</code></td>
</tr>
<tr>
<td><code>download_retro 0001000111000000</code></td>
</tr>
<tr>
<td><code>download_misr 0100011001100001</code></td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

STAT was used to evaluate different test solutions in terms of adopted TAM and test scheduling. As most of the cores are tested by means of dedicated BIST logic, the TAM bandwidth needed in this case is very low. For this reason, a first TAM selection consists in a standard IEEE 1500 serial line.

Figure 41 reports the metalanguage description file for the overall chip test structure, providing information about the TAM interconnection among the cores. The first three lines describe the implemented TAM (a single serial line, that is one group) and the power consumption constraint. Then, after the header GROUPS, the cores are enumerated including the name of their test description file and the estimated power consumption.
Chapter 3. The proposed IEEE 1500 SECT compliant test flow

A set of possible test structures and strategies has been evaluated by employing the STAT features, with one strictly defined user constraint: the processor cores tests, which utilize a functional approach, need to be run after the code memory test execution (cores 1 and 5). In this specific case, as forecast, no advantage was given by using a wider test bus or a more complex test structure; anyway, by carefully analyzing the different possible implementations of the serial TAM, it was possible to choose a serial line subdivided in two groups for reducing the total chain length. Suitable TAP instructions address one or the other group by controlling the switching logic that can be seen in figure 39.

While the ATE frequency is low (50 MHz), the frequency used by the BIST structures selected for this experimental case of study is higher and corresponds, for each core, to the maximum mission mode frequency. In the current case a common testing frequency of 250 MHz was employed.

Table XVII presents a summary of the SoC cores with power consumption (percentage wrt maximum circuit power) and test time corresponding to the selected test strategies.

### Table XVII

<table>
<thead>
<tr>
<th>SERIAL TAP</th>
<th>GROUPS</th>
<th>POWER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GROUP</th>
<th>00 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE</td>
<td>00 logic_1_pattern1.txt 100</td>
</tr>
<tr>
<td>CORE</td>
<td>01 mc8051_Aiip.txt 400</td>
</tr>
<tr>
<td>CORE</td>
<td>02 mem_1.txt 100</td>
</tr>
<tr>
<td>CORE</td>
<td>03 mem_1.txt 100</td>
</tr>
<tr>
<td>CORE</td>
<td>04 mem_1.txt 100</td>
</tr>
<tr>
<td>CORE</td>
<td>05 mem_1.txt 100</td>
</tr>
<tr>
<td>CORE</td>
<td>06 mem_2_mpblast.txt 350</td>
</tr>
<tr>
<td>CORE</td>
<td>07 mem_3.txt 200</td>
</tr>
<tr>
<td>CORE</td>
<td>08 logic2_pattern2.txt 100</td>
</tr>
<tr>
<td>CORE</td>
<td>09 mc8051iip_B.txt 400</td>
</tr>
</tbody>
</table>

---

Fig. 41: description of the SoC test architecture and TAM at system level
Tab. XVII: the SoC cores test strategies, power consumption and test time

<table>
<thead>
<tr>
<th>Core</th>
<th>Test structure</th>
<th>Power [%]</th>
<th>Test time</th>
</tr>
</thead>
<tbody>
<tr>
<td>X – logic</td>
<td>logic BIST</td>
<td>10.0</td>
<td>5.1 µs</td>
</tr>
<tr>
<td>Y – logic</td>
<td>logic BIST</td>
<td>10.0</td>
<td>4.3 µs</td>
</tr>
<tr>
<td>Memory 1-4</td>
<td>hardwired BIST</td>
<td>10.0</td>
<td>3.93 ms</td>
</tr>
<tr>
<td>Memory 5</td>
<td>programmable BIST</td>
<td>30.0</td>
<td>2.62 ms</td>
</tr>
<tr>
<td>Memory 6</td>
<td>programmable BIST</td>
<td>20.0</td>
<td>0.59 ms</td>
</tr>
<tr>
<td>µP A</td>
<td>SBST scan</td>
<td>40.0</td>
<td>25.3 ms, 931 µs</td>
</tr>
<tr>
<td>µP B</td>
<td>SBST scan</td>
<td>40.0</td>
<td>25.3 ms, 931 µs</td>
</tr>
</tbody>
</table>

As the reader can see from the table above, the processors tests are the most time-consuming among the considered cores. For this reason the test final schedule is forced to begin with the test of Memory 1 and 5 and µP A and B. The other core tests come in parallel along with the processor test whenever the TAM is free allowing data transfer to/from them.

The total SoC test time is 29.2 ms (29.5 ms when using a single serial chain), as shown in the timed graph reported in figure 42. STAT helps in finding a final implementation of the chosen test strategy, and it produces the STIL test description. The generated STIL program is about 107.4 Kbytes (2.7 Mbytes without macro compression).
Resorting to the STAT evaluating possibilities, a solution that includes the scan test vectors application to the microcontroller cores does not increase the total test time, due to the schedule flexibility of the considered system, but it raises the ATE memory occupation by 94.77 Kbytes.

Elsewhere, a manual writing of the whole test program took about ten days of work; this was due to the implementation complexity of the data transfer protocols and to multiple revisions that were needed to get it running. These time-consuming operations are no longer needed with the introduction of STAT: the test time evaluation and the generation of the generated STIL data need only the high-level description of the test structure at the core and system levels.

The produced STIL files have been transferred successfully to an industrial environment, exploiting an Agilent 93000 series ATE and a Smart Test data conversion.

The computation time of the test time evaluation and STIL program generation was 0.16 µs on a SunUltra 250 workstation equipped with 2 Gbytes of RAM and running at 400 MHz.
Chapter 4. Concluding remarks

The goals achieved in this work involve several aspects of the manufacturing SoC test. Principally, I focused on the definition of a common test interface, flexible enough to handle a set of DfT structures and permit easy diagnosis procedure management. I distinguished the following cases and, for each of them, studied a viable and cheap diagnosis-oriented solution at the core test layer:

- Memory cores [17][89][90][91]
- Processor cores [28][63]
- User-defined Logic cores [68].

The studied test interface is compliant with the IEEE 1500 SECT and guarantees a common test data communication inside the SoC structure. This test interface, usually referred to as wrapper, is the base for the system test layer techniques developed, including a diagnosis-oriented TAM and a Test Controller for memory cores [92][93].

The compliance with the IEEE 1500 SECT of the proposed core and system test layer structures finally allowed the implementation of a software tool [94] dealing with the last test layer that is the test application layer. This tool automatically generates the overall SoC test description in a native ATE language, the IEEE 1450 STIL language, and it
estimates the test costs in the early design phases by receiving the core and system test layer description as an input.

Moreover, a tool able to isolate faults responsible for misbehaviors has been implemented, which achieves good quality/occupation ratio in creating fault dictionary [95].

An industrial case of study is finally proposed demonstrating the effectiveness of the described techniques for fast yield improvement support [96].
Appendix A. A tool for Fault Diagnosis and data storage minimization

Up to now, one of the major concerns for semiconductor industries consists in reducing as soon as possible the yield loss when manufacturing integrated circuits. For this reason, the determination of the relation between defects and faults in digital circuits is currently one of the most investigated topics. This process is commonly known as fault diagnosis and consists in developing techniques able to supply the failure analysis process.

Fault diagnosis processes are computationally hard and memory expensive: the generation of structures able to store all the information needed to individuate a fault is a deeply investigated subject. These structures are called fault dictionaries [69][70][71] and their generation consists in selecting, within the circuit faulty responses, a subset of information allowing fault diagnosis. Several decisional processes have been proposed in the recent past, leading to different dictionary organizations. Many approaches have been proposed [72] to compress the dictionary size resorting to encoding techniques exploiting the regularity of the different organizations.

The most used dictionary organizations store diagnostic data in suitable matrices, tables, lists and trees [70][73][74]. The quality of a dictionary organization is given by
the diagnostic expectation they afford: this measure is the average size of undistinguished fault classes over all faults. A full resolution fault dictionary is organized in such a way that no diagnostic information is lost during its generation. All the following state-of-the-art techniques permit generating full resolution dictionaries. Pomeranz and al. [69] introduced the concept of compact fault dictionaries and demonstrated that only a few of the available information is needed for fault diagnosis. Chess and al. [71] have proposed an efficient error-set organization including only failure information. Boppana et al. [70] used a labeled tree representation; the tree solution is very attractive as it keeps the fault universe diagnostic classification provided by the pattern set. By traversing the obtained diagnostic tree from its root to a leaf, the set of faults responsible for the faulty behavior of the circuit may be identified. Each leaf corresponds to a so-called equivalent class, which is the set of faults exactly producing the same faulty responses for every applied pattern.

In this appendix, it is shown that smaller fault dictionaries can be obtained for combinational and scan circuitries by suitably ordering the applied pattern set. This approach permits reducing the overall number of information required for individuating those faults responsible for a faulty behavior, still maintaining a full dictionary resolution: the diagnostic expectation of a pattern set is not modified when their order is altered [74]. The proposed technique is based on a tree-based fault dictionary representation developed by following some of the principles of the compact dictionary representation. The contribution of this paper consists in an algorithm able to manipulate such a tree-based fault dictionary and resulting in more effective sequences of patterns; considering that in a tree-based fault dictionary an equivalent class is individuated by traversing the diagnostic tree from its root to a leaf, the pursued goal is the minimization of these paths by means of pattern reordering.

From the industrial point of view, the advantage in using such ordered pattern is twofold:
Appendix A. A tool for Fault Diagnosis and data storage minimization

- It minimizes the number of information in the fault dictionary, and hence its size.
- It reduces the average duration of the diagnostic process.

A.1 Tree-based dictionary organization

It has been stated in [69] that the fault dictionary size is a function of three test parameters: the number of patterns \( t_n \) included in the test set \( T \), the number of faults \( f_m \) included in the fault list \( F \), and the width of the faulty output responses \( o_{j,i} \) included in the output set \( O \), where \( i \) is in the range 0 to \( n-1 \) and \( j \) is in the range 0 to \( m-1 \). The output response width depends on the number of primary outputs \( p_o \). The size of a fault dictionary can be expressed in terms of symbol stored: depending on the dictionary representation, a symbol corresponds to a unit of information (e.g., a fault or pattern identifier, an observed value on a primary output, a pass/fail information).

A full fault dictionary representation stores for each couple \((t_i,f_j)\) the complete output response \( o_{i,j} \). Even if no additional computation is requested other than fault simulation, the cost in terms of stored information becomes prohibitive for large circuits. A full dictionary requires storing \( n \times m \times p_o \) symbols, where \( p_o \) is the number of primary outputs of the circuits. Its cost in terms of fault simulation time is also very high, since every couple \((t_i,f_j)\) has to be simulated.

On the contrary, a pass/fail fault dictionary is a dictionary representation that stores only a bit of information for each couple \((t_i,f_j)\) into a \( n \times m \) matrix [73]: a 1 in the \((t_i,f_j)\) position of the dictionary means that the fault \( f_j \) is detected by the pattern \( t_i \), while a 0 indicates that the pattern \( t_i \) fails in its detection. The regularity of pass/fail dictionaries and their relatively small size \((n \times m \text{ symbols})\) makes their use desirable, even if the diagnostic expectation value they provide could be lower than the value achieved by full fault dictionaries.

Assessing the ability of a pass/fail dictionary in diagnosing faults by adding only a few information about faulty responses is the key idea for compact dictionary generation.
Appendix A. A tool for Fault Diagnosis and data storage minimization

[69]: faults still not distinguished by a pass/fail dictionary are considered, and their output values are inserted in the pass/fail dictionary when the patterns outputs allow distinguishing new fault pairs.

The implemented tree-based representation follows the principles for generating compact dictionaries and requires two consecutive construction steps:

- **Step I**: a pass/fail binary tree is built to preliminarily group faults in equivalent classes determined using detection information, only.

- **Step II**: an output-based tree is built for each of those classes coming from step I, to further distinguish within their faults by observing faulty circuit responses.

Step I produces a first, still inaccurate fault classification by processing the coverage results, only. Every path traversing the resulting pass/fail binary tree from its root to a leaf identifies an equivalent class; these classes are called here coarse classes, as they can be possibly further divided by observing the faulty output responses of the circuit.

Step II takes advantage from this coarse classification, since the additional effort required to perform fault classification is reduced to discriminate between faults included in the coarse classes. Moreover, some of the coarse classes could be already constituted by a single fault, thus, not requiring any further inspection. Coarse classes analysis allows computing the final set of fine equivalent fault classes. This result is obtained by building an output-based diagnostic tree [70] for each of the coarse classes including multiple faults. Classification data are added to the binary tree structure each time an output response permits distinguishing between faults included in the same coarse fault class. This procedure aims at including in the fault dictionary only significant output values.

Three tables are required to store the obtained fault dictionary:

- **Coarse classes** table: it associates a number to a set of faults, whose equivalence is determined using the pass/fail binary tree.
Appendix A. A tool for Fault Diagnosis and data storage minimization

- **Pass/Fail** table: it stores the pass/fail sequence for each class included in the coarse classes table. A drop-when-distinguished approach is used to stop the in-depth construction of a tree branch when a leaf holds only 1 fault. Let’s define a detection string \( d_{si} \), which is the unique binary string isolating the \( i^{th} \) coarse class in the Pass/Fail table.

- **Fine classes** table: for each coarse class \( c_{cj} \) in the \( CC^* \) set of coarse classes including more than 1 fault, it itemizes the fine classes isolated by building output-based trees; each fine class is provided with the significant outputs that allow its separation from the other faults included in the coarse class it descends from. This information is stored resorting to a list-like representation: each line in this table includes the considered coarse class identifier and the determined subdivision in the fine class set \( FC \). For each fine class \( f_{cj} \), the significant output bits allowing its isolation are contained in the \( SO_j \) set and expressed using this notation: if the \( d^{th} \) output bit to test \( t_n \) and if the \( s^{th} \) output bit to test \( t_n \) are significant outputs for the fine class \( f_{cj} \), then the stored string will be \( f_{cj}:n,d:m,s; \).

<table>
<thead>
<tr>
<th>Tests</th>
<th>Fault-free output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_0 )</td>
<td>10010 00</td>
</tr>
<tr>
<td>( t_1 )</td>
<td>01111 00</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>11010 11</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>10101 11</td>
</tr>
</tbody>
</table>

Let’s consider as an example the test set proposed in [69] for the c17 circuit. Table XVIII and XIX contain the test set description and the full fault dictionary.

The binary tree representation obtained by executing the step I is shown in figure 43.a; a drop-when-distinguished approach is used when elaborating the pass/fail tree. Tableab. XX is the Coarse classes table, whose content refers to the binary tree leaves shown in figure 43.a; in table XXI it is reported the Pass/Fail table resulting from the
pass/fail binary tree: the first row lacks the last element, since fault $f_3$ can be distinguished by just observing the pass/fail information for the first 3 patterns.

<table>
<thead>
<tr>
<th></th>
<th>$o_0$</th>
<th>$o_1$</th>
<th>$o_2$</th>
<th>$o_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>00</td>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>$f_1$</td>
<td>11</td>
<td>00</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>$f_2$</td>
<td>00</td>
<td>11</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>$f_3$</td>
<td>10</td>
<td>00</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>$f_4$</td>
<td>10</td>
<td>00</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>$f_5$</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>$f_6$</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>$f_7$</td>
<td>01</td>
<td>00</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>$f_8$</td>
<td>00</td>
<td>00</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>$f_9$</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 43.b shows the two output-based trees required to divide the two coarse classes ($cc_1$ and $cc_6$) containing more than 1 fault. From those output-based trees table XXII descends. For the $cc_1$, significant outputs are all by $t_0$: in particular, $fc_0$ (including $f_1$) is obtained if both outputs are failing; $fc_1$ (including $f_4$) is obtained if only the first output is failing; $fc_2$ (including $f_7$) is obtained by induction if none of the other fine classes is identified.

The adopted notation allows directly identifying the equivalent fault class responsible for a faulty behavior by performing a reduced number of comparisons. In particular, decoding the stored information consists in:
Appendix A. A tool for Fault Diagnosis and data storage minimization

- Finding the observed pass/fail sequence in the Pass/Fail table, since each sequence in this table uniquely identifies a coarse class.

- In case the coarse class isolated contains more than one fault, comparing the significant outputs stored in the Fine classes table with the observed ones.

Figure 43: in a) the binary tree obtained by processing the pass/fail information is shown; in b) the output based tree is reported for the coarse classes larger than 1 fault existing in the binary tree.

Tab. XX: Coarse classes table.

<table>
<thead>
<tr>
<th>Coarse class</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent faults</td>
<td>3</td>
<td>1:4:7</td>
<td>2</td>
<td>0</td>
<td>9</td>
<td>5</td>
<td>6:8</td>
</tr>
</tbody>
</table>

Tab. XXI: Pass/Fail table.

<table>
<thead>
<tr>
<th>Coarse class</th>
<th>t₀</th>
<th>t₁</th>
<th>T₂</th>
<th>t₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Appendix A. A tool for Fault Diagnosis and data storage minimization

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Tab. XXII: Fine classes table.

<table>
<thead>
<tr>
<th>Fine</th>
<th>Coarse</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0</td>
<td>0:0</td>
<td>1:0</td>
<td>2:</td>
</tr>
<tr>
<td>6:3</td>
<td>0:0</td>
<td>1:0</td>
<td></td>
</tr>
</tbody>
</table>

A.2 Dictionary minimization

The technique aims at reducing the overall amount of information to be stored in a fault dictionary. The reduction is achieved by manipulating its tree representation: the detailed approach deals with the reduction of the information stored in the pass/fail binary tree, which is the structure required to store most of the diagnostic data in the tree-based dictionary organization illustrated in section A.1.

The technique detailed in the following paragraphs is based on the consideration that, in the pass/fail tree, faults are diagnosed by traversing the tree from its root to a leaf. The shorter these paths are, the smaller will be the pass/fail table in the fault dictionary.

The proposed algorithm exploits the properties of the tree for selecting a patterns order that minimizes the path length in the binary tree structure.

The proposed algorithm consists in the application of two procedures:

- A *dropping* procedure, performing a static length reduction of the tree branches.
- An *ordering* procedure, able to efficiently reorganize the tree structure to move leaves as close as possible to the tree root.

These procedures act on the Pass/Fail table and permit reducing its size without impacting the diagnostic expectation of the dictionary.
A.1.1 Dropping procedure

The commonly employed drop-when-distinguished procedure consists in eliminating a fault from the fault list when it has been isolated from all the other ones before definitively finishing the dictionary generation. Such a technique is useful as it permits reducing the number of simulations to be performed and the amount of information to be stored for diagnosing those faults completely distinguished by the pattern set. However, this technique is not effective for leaves corresponding to equivalent classes including more than one fault: paths leading to such leaves have to be completely built to prove that none of the used patterns is able to distinguish the contained faults.

The introduced procedure tries to reduce in length these tree branches by working directly on the pass/fail binary tree. Each branch leading to a multiple equivalent class is investigated and leaves reached by such paths are possibly moved up to previous tree levels. This tree modification is applicable when the leaf father has 1 child only, thus not providing any additional diagnostic information. This operation must not be performed if the tree edge to be eliminated is the only “fail” edge in the root-leaf path.

![Figure 44: the binary tree obtained by applying the detailed dropping procedure.](image-url)

An example of application of this procedure is in figure 44. When comparing this tree with the one in figure 43.a, the reader can note that the coarse class $cc_1$ has been moved up by 1 level in the tree, as no diagnostic information was provided by $t_3$. On the other
Appendix A. A tool for Fault Diagnosis and data storage minimization

side, coarse class \( cc_6 \) could not be moved up, as the faults it contains are detected only by \( t_3 \). The diagnostic expectation of the tree is not modified, while one symbol is eliminated from line 1 of the Pass/Fail table reported in table XXI.

A.1.2 Ordering procedure

The diagnostic expectation value offered by a test set is left unaltered when applying its patterns in a different order [74]. Let’s consider a fault list \( F \) including 4 faults and a test set \( T \) including 2 patterns \( t_0 \) and \( t_1 \), detecting \( f_0 \) and \( f_2 \), respectively. The diagnostic binary tree obtained by applying \( t_0 \) before \( t_1 \) is reported in figure 45.a. On the contrary, figure 45.b shows the binary tree built by applying \( t_1 \) before \( t_0 \).

![Figure 45: in a) and b) the binary tree obtained by applying the test set in its original and inverted order, respectively. The diagnostic expectation value is the same in a) and b).](image)

Leaves produced by these two different sequences are the same, apart from their locations in the tree representation: ‘central’ leaves are inverted, as they contain those faults alternately detected by the two tests.

These considerations can be easily moved to trees composed of more than two levels. The inversion of two patterns inside a test set modifies the tree structure in the following manner:

- Tree nodes generated by patterns preceding the inverted ones are left completely unchanged.
Tree nodes generated by patterns following those inverted are left with their content unmodified, but could occupy different positions, depending on the inverted patterns.

From the tree size point of view, pattern inversion is advantageous if it is able to move forward in the test set a pattern useless for fault diagnosis. As an example, let’s consider the tree represented in figure 44: $t_1$ does not provide any diagnostic information capable of dividing the fault set composed of $f_1, f_3, f_4, f_7$, while $t_2$ guarantees a distinction. Therefore, it is convenient to move $t_2$ before $t_1$. The result of $t_1/t_2$ inversion is reported in figure 46: $cc_0$ and $cc_1$ are now distinguished by applying 2 patterns instead of 3, thus allowing to delete 2 more symbols from the Pass/Fail table.

Nodes having only one child that generates two leaves are called here weak nodes and their localization in the tree is one of the key points of the ordering procedure proposed. Since finding an optimal pattern sequence is a NP-hard problem, a greedy algorithm is proposed, whose pseudo-code is shown in figure 47, where:

- The `min_weak_search(i)` function searches into the tree the weak node that is closer to the root; nodes belonging to levels from 0 to $i$ of the tree are not
investigated, that is the weak node has to be a node in a tree level greater than $i$: the returned value $n$ is the tree level in which the selected weak node is. If no weak nodes are found, the value 0 is returned.

- The $\text{invert}(i)$ function executes the tree transformation by means of exchanging pattern $i+1$ with pattern $i+2$.

- The $\text{drop_tree}(i)$ function allows applying the dropping rule introduced in paragraph A.1.1 starting from the $i^{th}$ level of the tree down.

The proposed algorithm starts by searching the weak node closer to the tree root and stores in the $n$ variable the level this node belongs to. If no weak nodes are found, it returns the 0 value; otherwise, a loop is entered and the following steps performed:

1. the pattern $t_n$ (generating nodes of the $n+1$ level) and $t_{n+1}$ (generating nodes of the $n+2$ level) are inverted.

2. tree branches length is minimized by applying the dropping procedure to those nodes potentially modified, i.e., those belonging to levels stemming from the level $n$.

3. a new weak node is searched in levels whose index is higher or equal to $n-1$. The $n-1$ level is selected instead of $n$ as new weak nodes could have been generated by the $\text{invert}()$ function in the $n-1$ level.

4. the process is repeated until no weak nodes are found, taking into account that a weak node cannot be processed twice.
\texttt{n = min\_weak\_search(0); while (n!=0) \\
\quad \{ invert(n); drop\_tree(n); \}}
\texttt{n = min\_weak\_search(n-1);}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{appendix_a.png}
\caption{pseudo-code of the greedy algorithm for pattern ordering.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{final_tree.png}
\caption{final form of the tree after pattern ordering.}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{c|cccc}
\hline
\textbf{Coarse Class} & \textbf{t₀} & \textbf{t₁} & \textbf{t₃} & \textbf{t₄} \\
\hline
0 & 1 & 1 & & \\
1 & 1 & 0 & & \\
2 & 0 & 0 & 1 & 1 \\
3 & 0 & 0 & 0 & 1 \\
4 & 0 & 1 & 1 & \\
5 & 0 & 1 & 0 & \\
6 & 0 & 0 & 1 & 0 \\
\hline
\end{tabular}
\caption{Pass/Fail table for the ordered patterns.}
\end{table}

For the considered example, a node can be referred in terms of the level it belongs and the ordinal position in this level starting from left. The first weak node selected is the node 1 in level 1: patterns \( t₁ \) and \( t₂ \) are inverted resulting in the tree shown in figure.
Appendix A. A tool for Fault Diagnosis and data storage minimization

46. The next weak node individuated is the node 3 in level 2: \( t_1 \) and \( t_3 \) are inverted resulting, after applying the dropping procedure, to the tree shown in figure 48. No other weak nodes can be found in the tree.

The minimized Pass/Fail table for the c17 example is reported in table XXIII. The number of stored symbols included in table XXI was initially 27; after the application of the algorithm, this value is reduced to 23.

A.3 Experimental Results

In this section, the results obtained on a benchmark set including some of the largest iscas-89 circuits and the itc-99 circuits are shown. All of them have been synthesized using a generic library developed in the Politecnico di Torino and are fully scanned. Test patterns for single stuck-at-faults have been generated using a commercial tool (Synopsys Tetramax).

Table XXIV summarizes the benchmarks general characteristics, in terms of number of faults included in the collapsed fault list, scan chain length, number of patterns, fault coverage and diagnostic expectation (DE) provided by the used patterns. The low values of DE are computed working on the collapsed fault lists, and show the good diagnostic quality of the used patterns. The selected set includes a variety of circuits with different diagnostic characteristics: beyond the number of faults to be classified, the computational effort strictly depends on the number of patterns, that determines the tree depth, and the number of primary outputs, that impact on the simulation times. The more complex circuits considered are b22 (few less than 100k faults and about 1,400 patterns) and b17 (70k faults and more than 1,400 primary outputs).

For all the considered benchmarks, the presented diagnostic procedure has been executed on a Sparc SUN Blade I, equipped with 2 GB of RAM memory. A prototypical implementation of the proposed technique has been written in C language, resorting to about 1,500 code lines.

To calculate the generated dictionary size, each symbol included in the Coarse classes and Fine classes dictionary tables to be expressed in a 32 bits notation, while symbols in
the Pass/Fail dictionary are in a 2 bits format, since the only values admitted in such a
table are ‘0’, ‘1’, and ‘new line’. Table XXV shows the size in bytes required to store the
fault dictionary of the considered benchmark set. Table XXV is organized in 5 columns,
itemizing for every studied circuit the size in bytes of the Coarse classes table, Fine
classes table, and Pass/Fail table: for the Pass/Fail table, its size before and after pattern
ordering and dropping are both reported, together with the percent reduction.

For the smallest circuits (up to b13), the contribution of the three tables is
comparable, even if the Coarse classes and Fine classes tables are often larger than the
Pass/Fail table. The consequent reason mainly derives from the low number of applied
patterns; pass/fail trees built for these circuits have few levels, therefore isolating lots of
coarse classes including several faults. To finally reach the full dictionary resolution a lot
of output information is required, thus slightly mitigating the advantage of the pattern
ordering strategy for these circuits.
Appendix A. A tool for Fault Diagnosis and data storage minimization

On the contrary, for larger circuits (up to b22) the pass/fail table size increases with the number of applied patterns. The more single fault equivalent classes are isolated directly by the pass/fail tree, the more number of additional information to be included in the Fine classes table is reduced, so taking additionally advantage of the increased number of observable points. For these circuits, the ordering approach permits reducing significantly the Pass/Fail table, and consequently the dictionary size. About $12 \times 10^6$ symbols are saved in the post-ordering Pass/Fail table for the b22 circuits.

<table>
<thead>
<tr>
<th>circuit</th>
<th>Coarse classes table [byte]</th>
<th>Fine classes table [byte]</th>
<th>Pass/Fail table [byte]</th>
<th>∆ size [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>s13207</td>
<td>57.4K</td>
<td>83.7K</td>
<td>41.2K</td>
<td>26.94</td>
</tr>
<tr>
<td>s15850</td>
<td>67.6K</td>
<td>74.5K</td>
<td>52.8K</td>
<td>20.45</td>
</tr>
<tr>
<td>s35932</td>
<td>207.2K</td>
<td>162.8K</td>
<td>247.2K</td>
<td>22.66</td>
</tr>
<tr>
<td>s38584</td>
<td>198.9K</td>
<td>170.8K</td>
<td>229.9K</td>
<td>11.78</td>
</tr>
<tr>
<td>s38417</td>
<td>197.7K</td>
<td>181.6K</td>
<td>191.9K</td>
<td>17.93</td>
</tr>
<tr>
<td>b05</td>
<td>13.5K</td>
<td>57.6K</td>
<td>813</td>
<td>31.16</td>
</tr>
<tr>
<td>b06</td>
<td>1.1K</td>
<td>1.7K</td>
<td>288</td>
<td>9.72</td>
</tr>
<tr>
<td>b08</td>
<td>3.5K</td>
<td>3.2K</td>
<td>1.7K</td>
<td>23.53</td>
</tr>
<tr>
<td>b09</td>
<td>3.4K</td>
<td>3.8K</td>
<td>1.6K</td>
<td>12.50</td>
</tr>
<tr>
<td>b10</td>
<td>3.8K</td>
<td>3.5K</td>
<td>2.4K</td>
<td>20.83</td>
</tr>
<tr>
<td>b11</td>
<td>13.3K</td>
<td>18.4K</td>
<td>18.1K</td>
<td>35.91</td>
</tr>
<tr>
<td>b12</td>
<td>26.2K</td>
<td>24.4K</td>
<td>27.8K</td>
<td>23.02</td>
</tr>
<tr>
<td>b13</td>
<td>6.5K</td>
<td>136K</td>
<td>431.7K</td>
<td>21.29</td>
</tr>
<tr>
<td>b15</td>
<td>165.1K</td>
<td>136.1K</td>
<td>413.7K</td>
<td>17.86</td>
</tr>
<tr>
<td>b17</td>
<td>555.6K</td>
<td>556.5K</td>
<td>2.4M</td>
<td>25.00</td>
</tr>
<tr>
<td>b20</td>
<td>459.6K</td>
<td>537.0K</td>
<td>4.4M</td>
<td>40.91</td>
</tr>
<tr>
<td>b21</td>
<td>470.5K</td>
<td>477.2K</td>
<td>5.4M</td>
<td>33.34</td>
</tr>
<tr>
<td>b22</td>
<td>693.8K</td>
<td>683.7K</td>
<td>8.2M</td>
<td>36.58</td>
</tr>
</tbody>
</table>

To summarize, it can thus be stated that the impact of the technique is higher for larger circuits, when it allows a significant reduction in the amount of storage required to represent the fault dictionary information.
Fault simulation (FS) costs and computational times are finally reported in table XXVI that includes, for each circuit, the number of required fault simulations (one pattern, one fault), separated in pass/fail FS and fine FS, and the time required for the ordering algorithm application.

A pass/fail FS only indicates if a given fault is detected by a given pattern. When considering a fault list, pass/fail FSs suit to be done in parallel, returning the list of faults covered by a pattern. In the adopted environment, the pass/fail tree is progressively built by computing information obtained by parallel pass/fail FSs; after each simulation the pass/fail tree is updated, and those faults classified in single equivalent fault classes are dropped from the fault list to be simulated next.

A fine FS provides the faulty circuit response for a selected pattern given a detected fault. In the exploited diagnostic environment, each coarse class determined by the pass/fail analysis is separately investigated; the stored pass/fail information permits calculating faulty responses for patterns detecting the selected faults, only. The parallelization of this process lightens the computational effort and allows building the output-based tree progressively: it is updated during fine simulations, and allows dropping those faults finely classified in single fault equivalent classes.

FS results reported in table XXVI have been obtained using a commercial tool (Synopsys Tetramax): pass/fail FS required little cpu time, since they are performed in parallel and on a fault list that becomes shorter and shorter thanks to the dropping features adopted. However, no feature allowing performing Fine FS in parallel is included in the tool, thus making the fine FS process more time consuming.

The time required for applying the ordering algorithm is reported in the last column of the table; the required cpu time, although sometimes non-negligible, it seems acceptable, since it represents a one-time cost providing significant results not only in terms of saved memory space, but also in terms of reduced diagnosis time. The circuits diagnosed with a large pattern set required more time, since the proposed algorithm analyzes the tree level-by-level. In order to apply the ordering algorithm, a pass/fail tree
is stored completely in memory; because of the memory configuration of the used computer, the diagnostic procedure for b18 and b19 circuits has not been done.

Tab. XXVI: fault simulation and ordering time costs.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Pass/fail FS</th>
<th>Fine FS</th>
<th>Ordering algorithm [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[##]</td>
<td>cpu time</td>
<td>[##]</td>
</tr>
<tr>
<td>s13207</td>
<td>540,677</td>
<td>1.62s</td>
<td>4,671</td>
</tr>
<tr>
<td>s15850</td>
<td>495,390</td>
<td>1.40s</td>
<td>4,304</td>
</tr>
<tr>
<td>s35932</td>
<td>1,442,363</td>
<td>6.53s</td>
<td>11,262</td>
</tr>
<tr>
<td>s38417</td>
<td>1,757,340</td>
<td>7.37s</td>
<td>10,686</td>
</tr>
<tr>
<td>s38584</td>
<td>1,812,307</td>
<td>7.58s</td>
<td>9,559</td>
</tr>
<tr>
<td>b05</td>
<td>119,248</td>
<td>0.69s</td>
<td>3,654</td>
</tr>
<tr>
<td>b06</td>
<td>1,925</td>
<td>0.13s</td>
<td>96</td>
</tr>
<tr>
<td>b08</td>
<td>12,224</td>
<td>0.20s</td>
<td>219</td>
</tr>
<tr>
<td>b09</td>
<td>11,377</td>
<td>0.25s</td>
<td>220</td>
</tr>
<tr>
<td>b10</td>
<td>14,535</td>
<td>0.29s</td>
<td>204</td>
</tr>
<tr>
<td>b11</td>
<td>156,982</td>
<td>0.83s</td>
<td>1,172</td>
</tr>
<tr>
<td>b12</td>
<td>211,053</td>
<td>0.06s</td>
<td>1,252</td>
</tr>
<tr>
<td>b13</td>
<td>21,591</td>
<td>0.25s</td>
<td>372</td>
</tr>
<tr>
<td>b15</td>
<td>3,696,827</td>
<td>8.92s</td>
<td>8,410</td>
</tr>
<tr>
<td>b17</td>
<td>22,925,425</td>
<td>64.47s</td>
<td>34,797</td>
</tr>
<tr>
<td>b20</td>
<td>72,358,829</td>
<td>6.47m</td>
<td>22,121</td>
</tr>
<tr>
<td>b21</td>
<td>78,930,690</td>
<td>6.78m</td>
<td>24,530</td>
</tr>
<tr>
<td>b22</td>
<td>117,372,894</td>
<td>11.34m</td>
<td>33,868</td>
</tr>
</tbody>
</table>
Appendix B. An industrial case-of-study

The present appendix proposes a new technique for yield loss detection that investigates the effects of different design parameters on the yield, exploiting a diagnostic-based approach. The main novelty presented here is to adapt the design diversity technique [75] to the yield loss analysis problem, already used for hardware and software fault tolerance. The vehicle chip is implemented through multiple implementations (N-version design) of the same design. The lots under investigation are composed of multiple instances generated through different strategies developed within the design flow. Differently from previous approaches adopting special test structures [76], memories [77], mixed signal chip [78] and ring oscillators [79], a suitable SoC is defined as a vehicle satisfying requirements of heterogeneity and complexity; moreover, special attention has been paid to obtain high testability and diagnosability.

By resorting to the explained approach, the reached yield level is considered as a further technology characterization parameter, while commonly area occupation, power requirements and maximum supported frequency are the only ones taken into consideration. In practical terms, the predicted technology yield obtained by static library analysis processes is reconsidered taking into account the physical SoC constraints and, therefore, the more effective set of selected library components.
Section 2 describes in details the motivations of this work and the yield analysis environment adopted during the experimental phases; Section 3 describes the structure of the vehicle chip and its testing-diagnostic capabilities. Section 4 presents some experimental results and, finally, Section 5 draws the conclusions.

**B.1 Background and motivations**

Recently, diagnostic algorithms have advanced enough to accurately and precisely diagnose many types of faults, including stuck-at, transition, bridges, and net opens. Multiple faults can be pinpointed in most cases and the type of fault can also be inferred by analyzing the circuit behavior across many patterns.

Considering new technologies, faults are often not random, but they are caused by inherent defectivity, finally responsible for a small but systematic yield loss; defects due to cell design marginalities or certain fab process steps are common. In order to identifying cells having a disproportionately high number of associations with defect sites, a graphical technique has been proposed in [80]: this approach aims at identifying the reason for high occurrence of faults affecting library cells. This information can be extracted from the histogram plot of cells versus the number of times the cell is candidate for a fail. Since all the cells are not placed in the design the same number of times, logic cells are physically different in terms of area and density. A ‘weight’ is assigned to each cell component, and then this value has to be normalized to the cell area. If defects are random, the largest cells will naturally be more often the suspect ones. By normalizing with respect to area, yet another picture that underlines cells requiring further examination is obtained. When the diagnostic process does not return a single candidate, a probabilistic scheme to weight the candidates is exploited.

This flow, which facilitates the identification of the most critical cells, can be finally moved to ‘family’ classification. Each cell used in the design comes from a standard library and almost all libraries can be divided into some classes (‘families’) containing different cell versions to implement a logic function. A family class may be created with various criteria. For example, the same logic operation was selected to group together
Appendix B. An industrial case-of-study

cells with similar layout features. Using the results obtained for single cells, all candidates coming from cells belonging to the same family in the library are collected.

In this way, it is possible to observe if one particular family is suspected more often than another, and in such a case, to observe if there are one or more cells showing higher failure rates. A cell belonging to a particular family, showing a high failure rate could represent a layout or design marginality detectable by observing layout and doing some failure hypothesis. A family showing an unusually high occurrence rate could indicate a repetitive layout feature, representative of a process issue.

Clearly, the interpretation of these effects strictly depends on the process, and could vary from process to process.

Up to now, critical cells in the devices have been simply identified, but it is really important to distinguish the marginalities coming from design or process [81]. A simple way to proceed is to verify if the failures associated with a particular cell are related to a particular region of the circuit or not. To do that, three correlation criteria are employed:

- **correlation with IDDq test results**: IDDq test consists of several measurements, taken in correspondence to the IDDq strobe states indicated by an ATPG engine, targeting a pseudo-IDDq fault model.

- **correlation with optical inspection data**: electrical failures are correlated with suspect process steps; digital images are captured during silicon deposition of the wafers, then these images are compared against the expected layout to identify abnormalities, called defectivity data. The correlation step consists in translating the X, Y coordinates of the defects found with the diagnosis tool and overlaying these results with in-line inspection data maps, stored in a dedicated database.

- **pattern re-ordering and fast analysis**: the correlation between applied patterns and their usefulness in terms of number of detected faults.
Appendix B. An industrial case-of-study

By adopting the described procedure, based on the analysis of different critical manufacturing factors, a minimization of the time required to improve the yield of new technologies is obtained. The purpose of this paper is to apply these principles on a set of identical circuits; each of the circuits analyzed is manufactured using the same technology but different parameters. In this way, library criticalities of the inspected technologies are underlined and a set of yield effective components is selected between the whole developed set.

The characteristics of the investigated architecture are extensively detailed in the following sections.

B.2 The proposed approach

The main purpose of this work is to exasperate the occurrence of technology criticalities inside System-on-Chips manufacturing process, then to apply the yield analysis flow recalled in section 2 to isolate and classify silicon criticalities.

The proposed approach consists in manufacturing a Design for Manufacturing (DfM) SoC used as a vehicle for yield analysis of a technology since its early development phases; a methodology has been defined relying on these two principles:

- a defined SoC structure equipped with the test structures necessary to perform detailed and low cost diagnosis procedures is needed
- many instances of the same SoC architecture are included on the same wafer by varying several physical implementation parameters.
Appendix B. An industrial case-of-study

In order to ease the optimal identification of the cores to be embedded in the DfM SoC, the characteristics that these cores should necessarily satisfy are the following:

- they should be characterized by a high coverage in terms of DC and AC faults, and, more in general, by a high diagnosability
- they should be fully known in terms of internal architecture.
- they should avoid any critical design structure, in order not to mix design and manufacturability issues.
- they should be repeatable in different versions, e.g., stemming from the adoption of different cell libraries, different synthesis directives, and different routing constraints.

The final set of cores is composed of different versions of possible cores. This choice has to guarantee the highest possible diversity of involved manufacturing library components and parameters. This diversity is exploited in order to explore possible malicious effects introduced by the new emerging technologies. The list of basic cores includes the following categories:

- Processor cores
- Memory cores
- User Defined Cores.

B.2.1 The DfM SoC

Taking into account the defined constraints, the manufactured SoC is composed of three cores:

- an 8-bit microcontroller
- a 32Kx8 bit sized SRAM memory
- a 16x16 parallel multiplier.
Appendix B. An industrial case-of-study

This chip was meant to be highly testable and diagnosable, not caring about its functionalities. However, as depicted in figure 49, during its possible mission mode the μP core reads the program to be executed from an external RAM memory, it communicates with the outside through its parallel port and it is able to drive the multiplier for arithmetic computations.

It is possible to achieve high diagnosability for each of the cited components resorting to the following test structures:

- an **Infrastructure-IP (I-IP)** manages the test procedures execution of the processor core [81]
- a **programmable BIST (pBIST)** is exploited for the memory core [17]
- the user defined logic is equipped with a **parametric logic BIST (IBIST)** [68]
- additional **scan structures** are inserted in order to improve the observability and controllability of the final test.

The diagnostic process consists in several repetitions of the test procedure varying the value to be loaded into the programmable structure of each of the employed test structure. Each repetition is called ‘diagnostic step’ and the values loaded in the programmable structure are intended to be the step ‘parameters’. In particular, a parametric strategy is adopted where:

- a diagnostic step is executed and its results processed
- if required, a new diagnostic step is programmed, using parameters that coming from the previous steps.

The test applied by each of the test structures is performed at-speed while the communications with the ATE are performed at low speed, and are supported by a P1500 SECT based structure. This solution permits pin saving, as it gives full access to the test structure with as few as five signals, and allows describing the test flow in ATE independent languages (e.g., STIL, CTL); additionally, the cost of such a flow can be
Appendix B. An industrial case-of-study

estimated in the early phases of the project both in terms of time for test and ATE requirements.

![Diagram of DfM System on Chip](image)

**Fig. 49:** Conceptual view of the adopted test structure for the DfM System on Chip.

*Processor core*

Test and diagnostic procedures for the processor core exploit the internal execution of a suitably generated test program. The execution of such a test program, loaded from the outside by the ATE into a selected memory space, allows stimulating and observing many parts of the processor core and is launched exploiting processors features, like interrupt ports. Upload, start and result collection are performed by an I-IP [81]. The I-IP structure is reported in figure 50.

The I-IP is considered as an I/O device and it is connected both to the system bus and to the processor interrupt port. The I-IP is able to activate functional test procedures by making use of the processor functionalities:

- a test program is loaded into a code buffer or directly in a predefined memory space by the upload internal module
Appendix B. An industrial case-of-study

- the loaded test program is executed as soon as the TEST ACTIVATION module forces a transition on the processor core interrupt pin
- the results are collected in the RESULT module by a MISR module.

The diagnostic process consists in the execution of several test programs, each one investigating a precise functional part of the circuit [82] (e.g., IU, pipeline registers, etc.); these programs are generated adopting an evolutionary technique [83]. Additional scan chains are inserted in the design to improve its controllability and observability.

![Conceptual view of the adopted test structure for a processor core.](image)

**Fig. 50:** Conceptual view of the adopted test structure for a processor core.

**Memory core**

In order to diagnose the memory core included into the DfM SoC, a programmable BIST is employed. This architecture, detailed in [17], permits the application of word-oriented tests for memory (usually March) by loading a small internal SRAM code memory: a control unit fetches and decodes the instruction loaded in the code memory during the initialization phase; finally a memory adapter module applies test vectors to the DUT. The architecture is explained in figure 51.
Appendix B. An industrial case-of-study

This structure allows diagnostic investigation by two ways:

- repetitive execution of the same algorithm allows extracting the failure bitmap without aliasing
- execution of different memory test algorithms allows to discriminate between equivalent faults.

User defined logic core

Regarding the UDL logic, that is a 16x16 multiplier, a common BIST architecture has been exploited for glue logic [68] applying pseudorandom patterns. The structure of such test architecture is reported in figure 52.

The IBIST architecture allows applying a user-definable number of pseudo-random patterns according with a selected seed previously uploaded in the ALFSR register. Test procedures for the multiplier also rely on a RETRO register, inserted to add controllability and observability in the diagnosis process: its content can be programmed and read from the outside.
Exploiting the parameters of programmability of this architecture, it is possible to reduce the dimension of the equivalent fault classes; that means precise classification of each single faulty behavior achieved by applying iteratively different independent pattern sets.

B.2.2 Design for criticalities

Many instances of the same SoC design are manufactured, varying the following parameters:

- Library components
- Place and route chip optimization
- Synthesis parameters.

This approach suits to maximize the probabilities that a chip includes technology-related defects. In particular, a procedure addressing the identification of weakness in the manufacture of each component and in the assembly process has been implemented for SoCs including different functionalities and requirements.
More in details, it is proposed to replicate a full diagnosable system directly in wafer: such a methodology allows product engineers to evaluate the response of each replica, finally drawing conclusions about defects located in different implementations. This flow introduces a twofold advantage:

- it individuates criticalities descending from the use of a set of library components
- it highlights marginalities led by different place and route architectures.

Figure 53 shows the parameter variation in the manufacturing process for the DfM SoC.

![Diagram](image)

**Fig. 53**: The synthesis flow.

### B.3 Experimental results

A pilot scheme has been evaluated to proof the feasibility and the effectiveness of DfM flow described. The wafer containing many instances of the DfM SoC is going to be manufactured in a 90nm CMOS-derived technology.

The SoC components work at high frequency of 200 MHz and above: the test resources partitioning scheme allow the utilization of a low-cost DFT-oriented tester, which matches the performance requirements demanded by the chip.
B.3.1 Abilities and costs of the flow

The costs of the flow have been quantified as:

- the cost of the additional area required to integrate the overall test architecture
- the cost for the ATE procedures, mainly in terms of time to test.

Table XXVII summarizes the extra area costs of each module of the design: the final overhead is less than 3% of the SoC original size. If a unique scan chain is introduced to support the processor procedures, 569 cells are serially connected.

<table>
<thead>
<tr>
<th>Core</th>
<th>Additional structure</th>
<th>Size [#gates]</th>
</tr>
</thead>
<tbody>
<tr>
<td>µP Scan chains</td>
<td>I-IP 12,159</td>
<td></td>
</tr>
<tr>
<td>MEM pBIST</td>
<td></td>
<td>9,068</td>
</tr>
<tr>
<td>Multiplier 16x16 LBIST</td>
<td></td>
<td>15,837</td>
</tr>
<tr>
<td>Original SoC size</td>
<td>1,442,137</td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td></td>
<td>2.57 %</td>
</tr>
</tbody>
</table>

Table XXVIII reports the maximum coverage capabilities related to the stuck-at fault model of the test applied by each test component and the requirements in terms of test time: this table has been filled by considering the ATE working at a low frequency (50 MHz), while the BIST circuitry internally tests each core at the mission frequency (200 MHz). The most significant figure is the execution time for the processor test, which corresponds to 92.5% of the complete procedure: this part of the SoC test also includes the upload operation of the test program into the RAM module.
Appendix B. An industrial case-of-study

Tab. XXVIII: Test time

<table>
<thead>
<tr>
<th>Core</th>
<th>Test strategy</th>
<th>FC [%]</th>
<th>Time [#µs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>µP</td>
<td>SBST (883 words)</td>
<td>98.8</td>
<td>25,234</td>
</tr>
<tr>
<td></td>
<td>Scan chains (80 patterns)</td>
<td></td>
<td>931</td>
</tr>
<tr>
<td>MEM</td>
<td>24n March</td>
<td>SAF</td>
<td>2,037</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SOF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DRF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>UWF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WTF</td>
<td></td>
</tr>
<tr>
<td>Multiplier</td>
<td>10x1024 patterns</td>
<td>100</td>
<td>149</td>
</tr>
</tbody>
</table>

Overall test time: 27,272

By exploiting the multisite features provided by the used ATE, the test of 4 instances of the DfM SoC can be performed in parallel, introducing up to 10% of time overhead.

The characteristics of the DfM SoC allow to precisely pinpoint the failing parts of the circuits: by suitably programming the included test structures, it is possible to minimize the size of faults classes, thus to classify the occurred faults and to underline library marginalities related to the constructive process and SoC characteristics. For the specific case-study, the diagnostic results are reported in table XXIX for the microprocessor and the 16x16 multiplier: the shown figures support the adoption of the proposed DfM SoC, since small equivalent fault classes are individuated by using the test infrastructures included in the chip.

B.3.2 Manufacturing consideration

The wafer reticule includes different instances of the DfM SoC, in which the several parameters can be considered for implementing intentional technology variation and exploring design corners:
Appendix B. An industrial case-of-study

- topology of the standard cells
- synthesis directives
  - clock tree implementation
  - place and route
  - frequency
- Back-end directives

This design of experiment aims at achieving an instrument capable of producing controlled data.

At this purpose, efforts are spent to implement a diagnosis oriented test flows, which implements extensively the testing in operating conditions corners. Current-based testing is utilized as well, by means of dedicated off-chip current monitors [84]. The produced data could then be analyzed through available tools and flows [80].

Tab. XXIX: diagnostic results expressed in terms of equivalent class size occurrence: the largest class size for the processor is 17, for the multiplier is 8.

<table>
<thead>
<tr>
<th>Core</th>
<th>Faults [#]</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>&gt;8</th>
</tr>
</thead>
<tbody>
<tr>
<td>µP</td>
<td>39,586</td>
<td>26,631</td>
<td>2,576</td>
<td>943</td>
<td>913</td>
<td>119</td>
<td>93</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Multiplier 16x16</td>
<td>13,058</td>
<td>9,943</td>
<td>632</td>
<td>230</td>
<td>122</td>
<td>76</td>
<td>41</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
0. References

References

0. References


0. References


0. References


0. References


0. References