POLITECNICO DI TORINO

Master's Degree in Embedded Systems

Master's Thesis

Automatic Generation of Self-Test Functional Stress Software Programs for System-Level Test

In partnership with STMicroelectronics

Supervisors:
Prof. Sonza Reorda Matteo
Prof. Bernardi Paolo
Cosupervisors:
Dr. Restifo Marco
Dr. Cantoro Riccardo

Candidate:
Rovere Enrico
ID: S252783

October 2019

Abstract

The present work discusses a framework for improving System Level testability of an Automotive SoC exploiting the Evolutionary Optimizer microGP to maximize core switching activity. The present framework does not require any knowledge about the Device Under Test but the Instruction Set it supports and its HDL netlist. The hardware peripherals discussed in the present thesis can vary the DUT operating voltage, going below and over specification, in order to exacerbate latent delay faults. The test programs are meant to be executed in the Burn-In phase of the post-manufacturing testing, on a dedicated board capable of varying the supply voltage. Functional testing is proposed to be merged with traditional ATPG-based and BIST tests in a System-Level-Test framework, which was shown to further increase the effectiveness of post-manufacturing tests. The evolutionary algorithm optimizer, provided with a set of instructions and operands, evolves generations of candidate Assembly programs targeting a specific submodule of the System-on-Chip. Each program is then simulated and ranked to progressively increase the switching activity of the target module. Moreover, the evaluating algorithm exploits a graph-based representation of instructions to penalize individuals with a high percentage of Write-After-Write hazards, which prevent the detection of errors occurred during the execution of the test program. At the end of the program execution the register state is verified against the golden output, producing the test result. The most promising test programs were then tested on physically faulty chips, at varying levels of supply voltage. Experimental results were obtained in collaboration with STMicroelectronics, which provided all the hardware employed in the present research and assisted for the whole duration of the present thesis.

Contents

List of Tables

List of Figures

Rovere Enrico

Introduction

When it comes to electronic products, the consumer often assumes that manufacturers have perfected their products' yields, and the ever increasing affordability of such products appears to confirm this intuition. In reality, this assumption couldn't be farther from the truth: with the exponential increase in transistor count dictated by Moore's law, greater and greater complexity comes in both theoretical design evaluation and especially in post-manufacturing testing.

This thesis describes a novel technique in testing and stressing processor cores, discussing the advantages and shortcomings in approaching the issue from a functional standpoint.

Functional testing of complex digital systems is the process of verifying the correctness of all the known function of a given Device Under Test. Rather than emphasizing on a given physical or structural fault model, this approach employs only the Instruction Set Architecture of the device, along with its gate-level netlist. This test approach may be executed in self-test mode, in addition to the traditional post-manufacturing testing. This versatility is desirable in mission-critical application such as the one under examination in the present work.

Rather than having a test engineer carefully devise a test program aimed at stressing the device, with the risk of overlooking one or more submodules, an evolutionary algorithm is tasked with generating the programs in question. This approach moves the issue a step back: the solutions provided by the algorithm would only be as good as the tests evaluating them. To account for this issue, a specific evaluation algorithm was created, simultaneously maximizing both the switching activity and the test instruction observability.

The present thesis is structured as follows: Chapter 1 provides the reader with essential notions on the physical processes for IC manufacturing. Moreover, it also describes the structure of an evolutionary algorithm such as the one employed for test program generation. Chapter 1 also describes the state of the art when it comes to functional testing and self-test procedures. Chapter 2 outlines the proposed approach and the Device-Under-Test, along with the results of the evolutionary program generation. Chapter 3 describes the hardware employed and the software produced to put in place the testing procedures theorized, with greater emphasis on the hardware. Lastly, Chapter 4 concludes with suggestions for further exploration on the research topic, and comments on improvements that could have been included in the research.

Chapter 1

Background

The following chapter aims to describe the fundamental notions concerning the topics spanned by the present thesis. Initially the general steps in Integrated Circuits design and production are outlined, discussing the relevant electronic notions. Then, physical phenomena are characterized and identified as fault models, introducing testing steps put in place to detect and discard malfunctioning devices. An overview of the concepts of Functional Testing and Software-Based Self-Test is also provided, along with an introduction to Evolutionary Algorithms.

1.1 Electronics

1.1.1 Integrated Circuit design

1. **Specification** · The designer determines the expected behaviour of the device.

2. **Design** · An Hardware Description is produced, in a suitable HDL language such as VHDL or Verilog.

3. **Simulation** · The resulting entity is then simulated and verified to be consistent with the implementation requirements.

4. **Synthesis** · Using a software tool such as Synopsys' Design Compiler and a library of logic gates, the Hardware Description is synthesised into a "netlist".

5. **Verification** · The netlist, which is meant to fully reflect the designer's HDL description, is verified to still respect the design constraints.

6. **Place and Route** · A software tool such as Cadence's SoC Encounter performs the so-called "Place and Route" process, which maps the synthesised logic gates with physical features and routes the circuit interconnects.

7. **Manufacturing** · Once the physical design is proven to be compliant, the ”floorplan” - the physical counterpart of the netlist - is supplied to a semiconductor manufacturing company, which produces the IC silicon and packages it.

1.1.2 Photolithography

Photolithography is the most widely used process for producing Integrated Circuits; it employs UV light to sequentially transfer a geometric pattern from a series of optical masks onto a ”wafer”, a slice of monocrystalline silicon ingot. This collection of quartz masks represents a series of horizontal cross sections of the device's floorplan, the role of which is outlined in Section 1.1.1.

As depicted in Figure 1.1, first the photoresist, an insoluble substance that becomes soluble when exposed to UV light, is deposed in a thin film on the wafer surface. Light is then shone on the film through the mask and washed to remove the unwanted regions, leaving a portion of the mask in place so that etching of doping can be performed. A special solvent can finally remove the photoresist pattern, and the cycle is repeated with the following mask until the entire microstructure is complete.

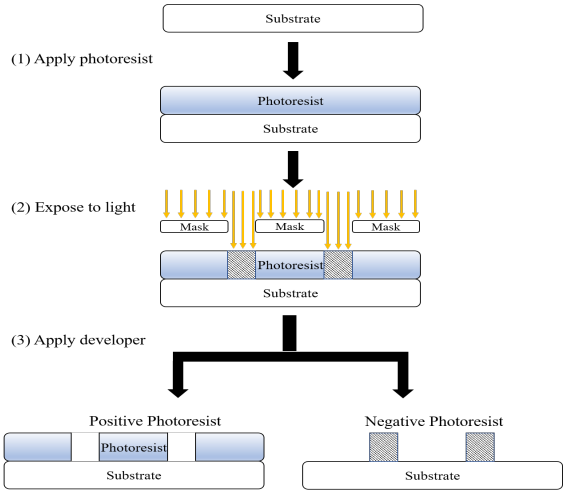

Figure 1.1: A step of the photol. process

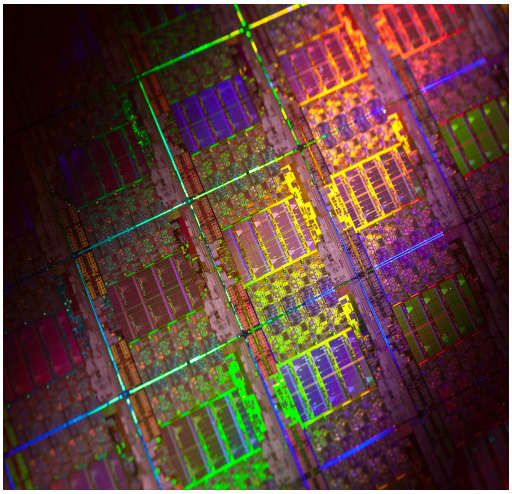

Figure 1.2: Detail of a silicon wafer

Manufacturing processes are classified by a metric named *feature size*, meaning the unit length of that process. A typical automotive manufacturing node would be 45 nm, while a more cutting-edge process could scale as down as 7 nm, employing advanced tools such as Extreme Ultra Violet (EUV) light sources and multiple masks to produce a single layer. [23]

1.1.3 Packaging

At this point the Integrated Circuit is complete and can be preliminarily tested, by means of a process outlined in Section **??**. However, since silicon devices are very susceptible to mechanical stress and chemical contamination, not to mention overheating, they are enclosed in a epoxy or ceramic case, denominated *package*. A package provides pins or leads interfacing the die with the operating environment, in addition to supplying power to the device.

Moreover, in System-on-Chips more than one device is installed in a single package, requiring further die-to-die routing, which is accomplished by means of bond wires.

Bond wires

Bond wires are extremely thin strands of metal or alloy connecting a device to package pins or to another die in SoCs. Figures 1.3 and 1.4 show the ultrasonic or spot welding employed to provide a solid connection between wire and die, but this process is extremely critical and susceptible to shearing and die delamination [9].

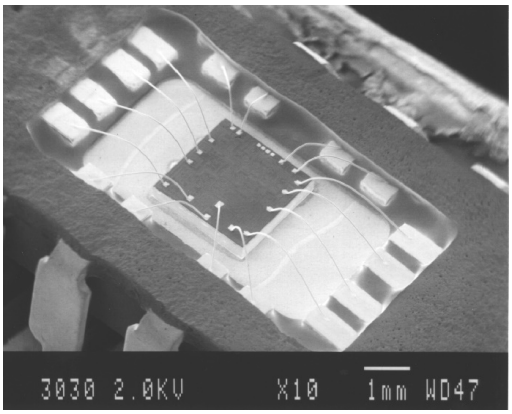

Figure 1.3: A packaged die exposing its bond wires [9]

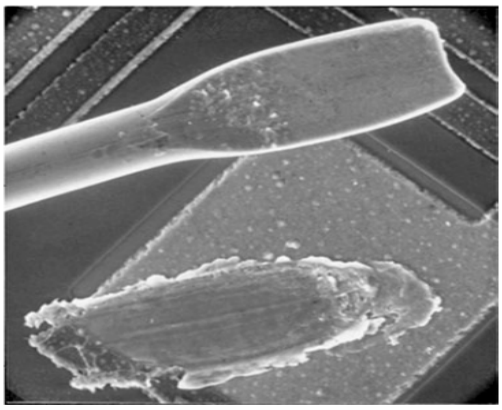

Figure 1.4: Detail of a wire sheared off its pad [9]

Bond wire interconnects may not fail as dramatically as depicted in Figure 1.4, but rather more subtly, in a change in the interconnect parameters. Specifically, the DC characteristics could be affected, increasing the IR drop through the line, or the frequency response may vary, possibly leading signal losses or to critical timing violations in the system.

Dynamic Power

Power dissipation in CMOS circuits has two components: static, due to leakage current, which in turn is dependent on the technology node and increases the smaller it goes; and dynamic, due to switching activity. The static power is relatively low and is often neglected in power estimation. In Figure 1.5 the green line shows the path of the static (leakage) power, flowing irrespectively of the logic value of the gate. The blue line is instead the dynamic power, flowing when the output is toggled.

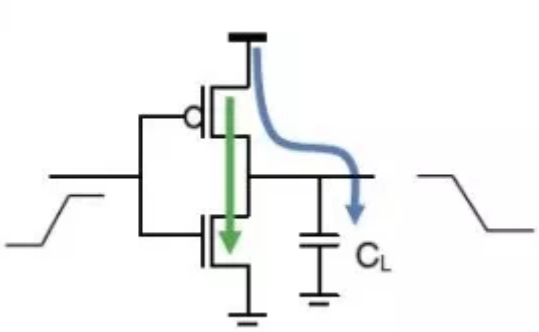

Figure 1.5: An inverter switching

Once the processing and structural parameters have been fixed, the measure of power dissipation is dominated by the switching activity (toggle counts) of the circuit. Achieving a peak power draw condition involves maximization of a circuit's switching function, that is the toggle count. The following formula shows the mathematical relation[11] between switching activity and dynamic power:

$$P = P_{leak} + P_{dyn}$$

$$P_{dyn} = \frac{1}{2} \cdot V_{DD} \cdot f_{clk} \cdot C_L \cdot E_{SW}$$

The last factor E_{SW} is the *switching activity*, which as one can see has a linear relation to switching power.

1.1.4 Process variability

The continually shrinking of die sizes and the consecutive complication in manufacturing such devices brings about a more and more significant factor in SoC failure, that is *process variability*. This phenomenon is the variation of the geometric attributes of transistors on the die, such as width or thickness. Since the feature size of modern devices is of nanometric scale, this variance may lead to performance degradation, and the device may behave differently than simulated.

The effect of such variation may manifest in what's called a *defect*, the type of which may be grouped [7] in two macroclasses:

1. **Patent defect** · A condition which does not meet specifications and is readily detectable by inspection or testing.
2. **Latent defect** · A defect that is not detectable by inspection or functional testing until it is transformed into a patent defect by environmental stresses applied over time.

Figures 1.6 and 1.7 show two different types of patent defects, the former reducing the ohmic value of a MOS gate and increasing its quiescent current, the latter altering the mutual capacitance of lines A to C and possibly increasing the crosstalk between the two.

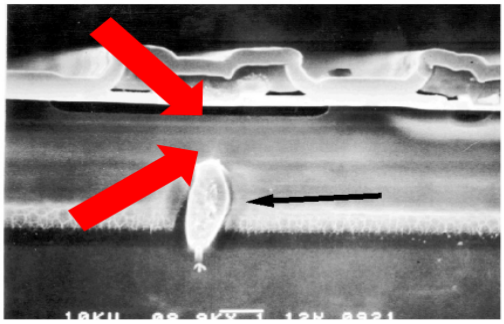

Figure 1.6: An impurity (black arrow) in a MOS gate (red arrows) [18]

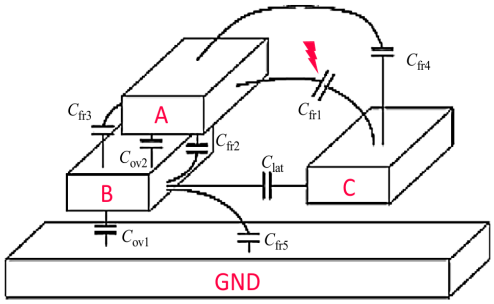

Figure 1.7: Schematic of interconnect capacitances [22]

A slight drift in the shape of a feature is named *marginality*[14].

1.2 Faults

As outlined in Section 1.1, a significant portion of devices leaving the production line are affected by some kind of latent defect. Over time, defects manifest in the form of *faults*, which may imply different logic misbehaviours. Fault models are therefore crucial to design testing procedures for digital systems. The most notable are:

Stuck-at · A given net is stuck to either HIGH or LOW logic values, possibly due to a short to a power rail.

Bridging fault · Two signals are shorted together, and the resulting behaviour depends on the nets' driving strength.

Transition delay · The parameters of a line are altered in such a way that signal propagation velocity is lowered, and may lead to logic timing violations.

1.2.1 Automatic Test Pattern Generation (ATPG)

ATPG is a "brute force" approach to Integrated Systems testing. Given the device's netlist, a Fault Simulator injects faults into it and attempts to find a *pattern* to excite the fault operating on the device's inputs and observing the outputs. Figure 1.8 shows a typical DUT on which test access logic such as ATEs would operate[20]. The output of the circuit is compared to a reference "golden circuit", with no faults present.

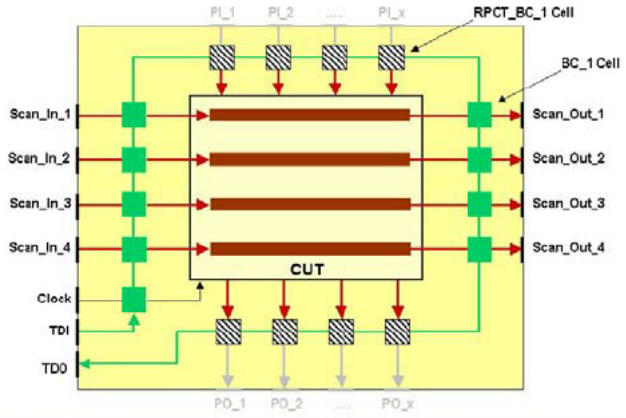

Figure 1.8: Primary and pseudoprimary I/O in a digital system

ATPGs produce what are called *patterns*, a set of input values to excite one or more faults. Modern devices can have thousands of inputs, and therefore producing and storing patterns is a very costly operation.

1.2.2 Dead-On-Arrival mitigation

All devices that pass post-manufacturing tests would be expected to function correctly. In reality, a significant amount of products shipped will be Dead-On-Arrival, that is not function correctly once in the consumer's hands. When the device would be retested by the manufacturer, often if would pass the tests again, creating what's called a No Failure Found (NFF) event.

When a part is shipped, all inputs applied to it, whether in a manufacturing or user environment, constitute an additional set of tests. These tests detect a subset of defects that are not identical to those found by IC level tests. The size of the intersection of these sets of defects is an important indicator of test quality. One can call the user test environment, which includes both system test and user applications, a field test.

A defect in the region of the board/system test or field test not covered by the manufacturing test causes a No Failure Found (NFF), and should not be considered anomalous. All defects occurring within the region covered by the IC test should have been removed. If a retest shows that a defect is in this region, the most likely explanation is that the defect occurred after the IC test was done.

An NFF component has the following characteristics[5]:

- It has passed a certain test step in the component lifecycle.
- It has failed a subsequent test, or has been indicted as the cause of failure of a higher level part of which it is a component.
- It passes the first test step again.

One example of an NFF is a microprocessor which passes all levels of IC test, is indicted as causing a failure during system test and upon return to the supplier, passes the IC test again. Another is a field replaceable unit (FRU) which passes individual board test and is shipped to the field, and which later is diagnosed as being faulty, but which passes board and system test at a repair depot.

The failure rate in the field is a measure for the dependability and reliability of systems, and well-specified targets have to be kept or improved. Diagnosis is mandatory, and in application fields like automotive or avionics the root cause of any system failure has to be clarified. Many failures are only observable under specific operating conditions, and they may disappear after disassembling the system., thus requiring for part of the system to be part of the diagnosing process. These "No Failure Found" (NFF) cases are expensive and introduce also risk for other products[12]. Moreover, the OEM (Original Equipment Manufacturer) relies on rather a long supply chain, and fault identification requires the collaboration of many partners. If boards and chips offered sufficient self-diagnosis capabilities to the OEM directly, it will help shortening the diagnosis process; this procedure may

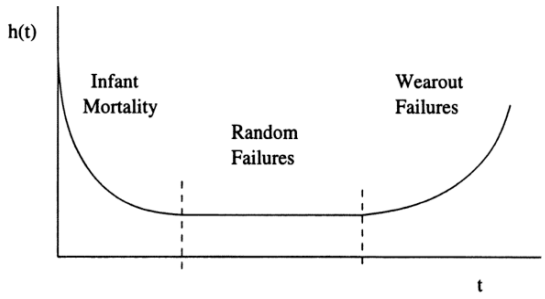

Figure 1.9: A typical bathtub curve

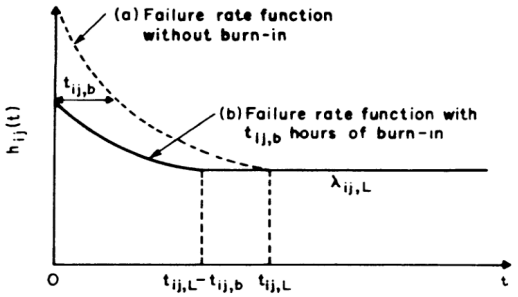

Figure 1.10: Burn-in mitigation of infant mortality [13]

be used under typical operating conditions to mitigate the NFFs before disassembling the system. The collected data will benefit not just the OEM but also all the members of the supply chain[12].

Statistics show a high failure rate in the early stages of components life, roughly following what's called a "bathtub curve", shown in Figure 1.9. As one can clearly see, the phenomenon of "infant mortality" is very much real, and must be accounted for in yield estimation. Several mitigation techniques are employed in order to mitigate it, such as Burn-in and ESS.

Burn-in

The purpose of the Burn-In (BI) process is to activate infant mortality (early life latent defects) that naturally affect populations of electronic devices. The Burn-In approach uses higher temperature and higher voltage than user mode to accelerate the early life phases of a product [19]. Burn-in (BI) is a manufacturing test phase used for any mission-critical product, such as the case study SoC discussed in Chapter 2.

The Testing Equipment applies two types of stress. The former is is the Electrical stress, which is produced by activating the functionalities of the devices during the BI phase; The latter is the Thermal stress, which is introduced by means of a climatic chamber which warms the chips up to their specification limits — this type of stress is directly related to Arrhenius's law about material aging. The main idea of BI is to combine thermal and electrical stress to accelerate the activation of extrinsic defects under the bathtub curve hypothesis.

Scan chains

Modern process technologies and design tools allow the realization of very large and complex systems on a single die. Because of the increased system complexity, verification techniques such as simulation, formal verification, static timing analysis,

and emulation cannot guarantee that first silicon is designed error free. Therefore, techniques are necessary to efficiently debug first-silicon[16]. If the input or output behavior of the chip is not correct, one must 'zoom in on the error' by accessing the signals and memories inside the chip. To enable this functionality, additional features must be added to the design. The flip-flops and embedded memories inside the chip can be made accessible through the scan chains, depicted in Figure 1.11.

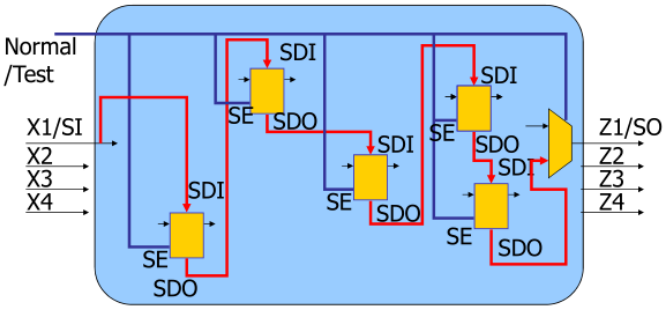

Figure 1.11: A diagram of the typical scan chain architecture

The standard Design-for-Testability (DfT) practice is to multiplex the scan I/O with the functional pins, as in Figure 1.11. However, when an IC is placed in its application environment for debug, these pins are no longer free for scanning. Therefore, making the chip fully scannable is not enough. To make the scan chains accessible in the application environment a scan chain wrapper is added during the design phase. This wrapper makes the scan chains serially accessible through the JTAG (Boundary Scan) port.

1.3 An evolutionary algorithm

An evolutionary algorithm (EA) is an algorithm inspired by the theory of evolution which claims that "animals and plants have their origin in other types, and that the distinguishable differences are due to modifications in successive generations" [21]. It could lay the foundations in the Darwinian concept of reproduction and the survival of the fittest.

Evolutionary computation is a field of computer science in charge of solving and developing these algorithms. In every evolutionary algorithm, a single candidate solution is named *individual* and the set of all individuals existing at a particular time is called population. It is also important to highlight that evolution proceeds through discrete steps called *generations*.

Although there are many different evolutionary algorithms, the common idea behind them is the same: given a population of individuals the environmental pressure causes natural selection (survival of the fittest) and this causes a rise in the fitness

of the population [6]. Over time, the best individuals will survive and evolve in artificially constrained environment, while the others will be discarded. This process is iterative and continues until a solution is found or a pre-set limit is reached.

Therefore, the final population will be entirely different from the initial random one. It will be composed of the fittest test programs, that can then be selected and cherry picked. An evolutionary algorithm is strongly inspired by biological mechanisms such as reproduction, mutation, recombination and selection. From this perspective, evolution is often seen as a process of adaptation and fitness function as an expression of environmental requirements and not as an objective function to be optimized. The role of a *fitness evaluation* in an evolutionary algorithm is to define how good the current solution is, determining the base for selection.

An evolutionary algorithm can evolve applying some operators like *recombination* or *mutation*. The former is applied to two or more randomly selected candidates (also called parents) and produces one or more candidates (offspring). When the offspring is aged sufficiently, the evolutionary algorithm selects the candidates for the next generation according to their fitness parameters. Notice that the entire process is ruled by two important forces that put the basis of an evolutionary system: the variation operators and the selection. The cooperation of the two leads to improving fitness values in the following populations [17]. This process can be represented as flow chart, shown in Figure 1.12

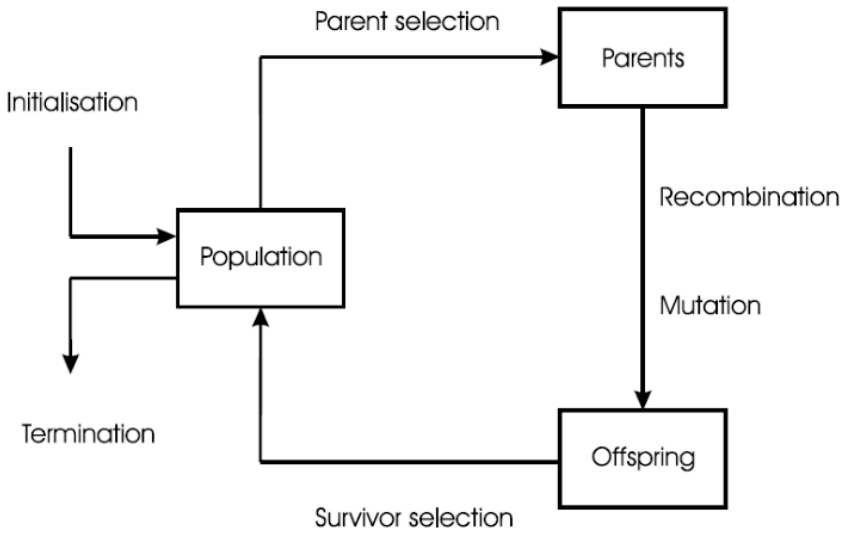

Figure 1.12: A general evolutionary algorithm as flow chart

1.3.1 Micro Genetic Programming (μGP)

Micro Genetic Programming (μGP) is an evolutionary algorithm able to find an optimal solution to hard problems. μGP was created by CAD Group at Politecnico di Torino in 2002. Although the initial idea was to generate assembly-language programs for testing different microprocessors, nowadays it was also employed in many other fields such as: creation of test programs for pre and post-silicon validation; design of Bayesian networks; creation of mathematical functions represented as trees; integer and combinatorial optimization.

Starting from an initial set of programs, also called individuals, μGP is capable of iteratively improve and evolve them, according to feedback metrics provided to the evolutionary core by the fitness function. Its heuristic algorithm uses the result of the evaluations, together with other internal information, to ex- plore the search space and to produce the optimal solution[21]. μGP algorithm can be modelled by the composition of three different blocks:

Evolutionary core · Computation and selection take place in this block: the population is the current group of candidate solutions managed by evolutionary core; at the beginning it is composed by random programs. The user can then select the initial size of the population which is greater or equal to the final one.

Instruction pool · Used to ensure individuals are valid assembly language programs[3]. It can be seen as a library including a highly concise description of the assembly syntax or parametric fragments of code. Arguments are generated by the evolutionary core.

Fitness evaluator · When an individual is ready, it is passed to an external evaluator, also referred to as *fitness function*, a tool provided by the user which simulates the program on an RTL simulator. It drives the optimization process taking an individual as input and producing a collection of numeric values as output. This collection is finally provided to the evolutionary core with the necessary feedback, closing the loop of the evolution. Its role is to define how "fit" the evaluated solution is with respect to the problem to solve.

The μGP algorithm selects the next generation, depending on their fitness values, favouring those with higher quality, although the concept of age is also frequently used. When μGP chooses parents for the new population, it randomly selects a given number of individuals and picks the best ones among them via tournament selection.

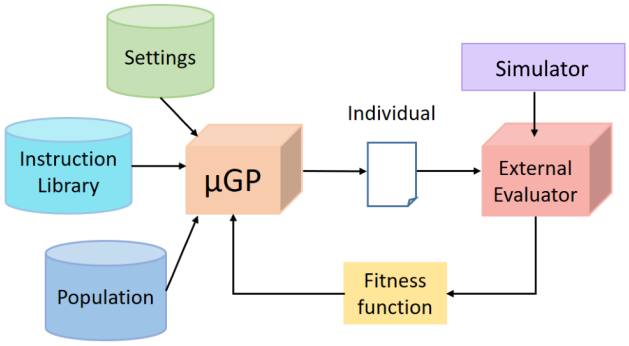

Figure 1.13: μGP flow chart schematic

In each generation, the algorithm generates offspring by applying different operators (*mutation* and *crossover*), fully self-adapting the strength of operators. At the end of the survivor selection mechanism, the old population is discarded and the new one is evaluated by the external evaluator, individual by individual. This process ends when a termination condition occurs:

1. Maximum CPU time reached.
2. Limit of evaluation is reached.
3. Fitness does not improve for a given amount of time.
4. Population diversity drops under a preset threshold.

These stopping conditions are mandatory, since the heuristic and iterative nature of evolutionary algorithms can't guarantee the optimization will reach an absolute maximum[17].

1.3.2 Individual representation

A major issue when devising a test program generator for microprocessors is how to represent test programs in a way that alows for efficient manipulation, while guaranteeing syntactical correctness [4].

Each test program, called individual, is internally represented as a Directed Acyclic Graph (DAG, shown in Figure 1.14) and must adhere to restricted rules. A DAG is *direct*, that is its edges are directional, and *acyclic*, that is no loops exist within the graph. Each node corresponds to a valid assembly instruction, where the syntax is defined in the Instruction pool and the operands are represented by the parameters associated to the node.

µGP allows the DAG to be include sub-DAGs of different type (procedures, traps, main program) called frames. Each node can have references to any other

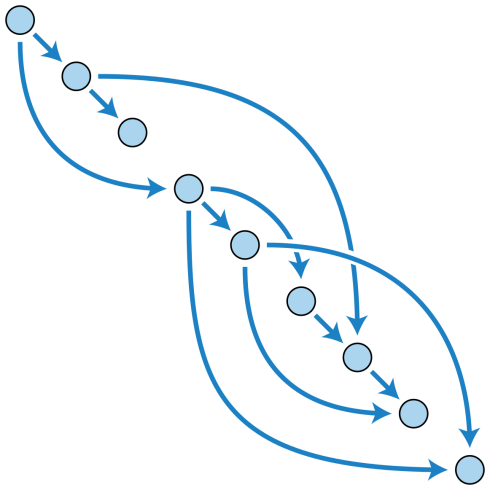

Figure 1.14: A Directed Acyclic Graph

node in its subgraph or to the beginning of a different subgraph[21]. A sub-DAG is composed by three types of nodes:

1. **Prologue/Epilogue** · These nodes are mandatory and must al- ways be present. They represent required operations, such as function declarations and initializations. The prologue is the first node of the program and has no parent nodes, while the epilogue is the last one and has no children. They depend both on the processor and on the operating environment and on the frame type, and they may be empty[4].

2. **Sequential Instruction** · These nodes represent common arithmetical or logical instructions. Due to conditional or unconditional branches, some of them could be unreachable.

3. **Conditional branch** · Includign all the conditional branches defined for the target assembly language and provided by the instruction pool.

In the present case study, only three nodes were employed, *Prologue*, *Body* and *Epilogue*, since conditional branches exploration was not a topic of interest. However, adding some could help target the control logic as well.

1.4 Functional testing

Functional testing of microprocessors and processor cores has been extensively studied the last decades. Functional testing does not try to obtain high coverage of a particular physical or structural fault model. It rather aims to test for the correctness of all known functions performed by the digital circuit. In the case of

processors, functional testing aims to cover the different functions implemented by the processor's instruction set, and for this type of circuits it seems to be a very "natural" choiee. Therefore, functional testing of processors needs only the instruction set architecture (ISA) information for the processor to develop test pattern sets for the processor testing and no other lower level (like gate level) model of the processor. Functional test sets may be applied either externally or internally in a self-test mode[8].

This test approach may be adopted in different system test scenarios and may be motivated by different reasons. When addressing board-level test, functional test is typically considered as the final step, which is supposed to complement the previous ones with specific goals (e.g., testing the interfaces), allowing to achieve the target defect coverage. Further examples of usage of functional test at the system level include the following[12] cases:

- During the manufacturing test of a System on Chip (SoC), functional test may complement structural test because it may cover some defects that are not detected by the latter, for instance because the former typically works at-speed (while some DfT techniques do not), or because the functional test exercises the system exactly in the same conditions of the operational environment.
- Before mounting a device on a board, it may be required by regulations or economically convenient to perform a test to check whether the device is fault free (independently on the test performed by the device manufacturer). This test - the *Incoming Inspection* - is performed by the customer company and it is often only based on a functional approach, typically because possible Design for Testability features are not documented by the device provider, and therefore not usable.
- During the in-field test of a board, it may happen that the DfT features of the composing devices are not accessible - may require an ATE - or are not documented by the device providers. Hence, the only feasible solution for the OEM company is often based on a functional test. If the device provider does not provide an effective test procedure, a functional test procedure has to be developed from scratch, exploiting the features of the device

The drawback of functional testing is that it is not directly connected to the actual structural testability of the processor, which is related to the physieal defects. The structural testability that a functional testing achieves strongly depends on the set of data (operands) which are used to test the functions of a processor. In most cases, pseudorandom operands are employed in functional testing and they lead to test sets or test programs with excessively large test application time not capable to reach high structural fault coverage. In the present case study, however, the test programs' operands are individually evolved to maximize the switching activity, hinting at greater test performance with respect to random operands.

Structural testing on the other side, targets a specific structural fault model. EDA tools can be used for automatie generation of test sequences (ATPG tools) with the possible support of structured DIT techniques like scan chains or test points insertion. However, the pattern-based automatic testing of digital circuits discussed in Section 1.2.1 is inadequate for several reasons. The large amount of logic - in the order of billions of gates - combined with the fact that SoCs may embed logic not easily testable, makes it impractical to use existing fault simulation and test generation software to derive tests.

Physical failures in integrated circuits result in logic behavior which cannot be modeled by existing faults[1]. Another approach to testing microprocessors is to use pseudorandom test patterns [6]. However, the problem with this approach is that the fault model for the instruction sequencing is not accurate in modeling the faults in the instruction execution process; moreover, the length of pattern-based tests is becoming larger and larger, and so is the memory required to store both input and output patterns.

In the framework of System-Level-Test functional testing is employed to complement ATPG pattern procedures, further stressing the Device Under Test (DUT). In various kinds of applications, especially in the present context of critical reliabiity, the consumer must ensure the correct functional operations of the purchased chips even not necessarily knowing their detailed implementation. Most of the test vectors were derived based only on certain sets of modelled faults (described in Section 1.2) and may not target test certain faults (e.g., bridging fault) in a comprehensive way. Another possible approach is to run a pseudorandom test sequence of test vectors for simplicity and minimum cost. For example, to test a microprocessor, test engineers may generate random streams of instructions and compare the results with a pretested ”golden” module. But, to derive reliable measures of fault coverage using this method is a nondeterministic problem.

To address the problem of high functional complexity of LSI/VLSI devices (e.g., microprocessors), ”Divide and Conquer” techniques (e.g., modular decomposition) are usually used to subdivide the devices into functional modules whose behaviors could be individually verified using aforementioned self-test procedures. For example, testing of a microprocessor can be subdivided into testing data processing and control modules. But two situations may often be observed:

- Testing is focused on data processing function, faults in the control part are usually simplified.
- Modules partitioning and tests are developed on an ad hoc basis, targeting specific entities.

The order of test sequence is of major concern during functional testing. It is usually derived from the functional description of the device (e.g., observability and controllability of function outputs and inputs) and associated parameters (e.g., type

of functional fault under test). In the present case, the order is determined by the existing Burn-In procedure, using Functional Testing to verify and further stress the device after the ATPG pattern application, Figure 1.15.

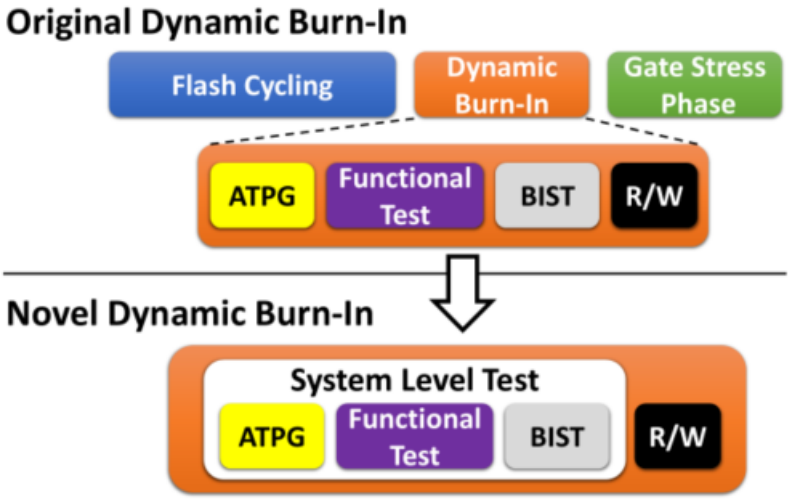

Figure 1.15: Proposed test flow

The proposed modification to the test plan restructures the order of the test phases in the following[19] manner:

The ATPG phase · configuring all the scan chains of the DUT into a single scan chain. This configuration is called "burn-in scan chain" configuration. The tester accesses the resulting burn-in scan chain and performs a toggling activity promotion. This phase stresses the DUT uniformly.

The Functional Test phase · the ATE loads the functional test program into the SoC Flash memory, then the tester triggers the execution of the self-test program and downloads the results of the functional test. The program can correspond to a procedure for stressing the DUT, like the common Functional Test phase of the Dynamic Burn-In, or a procedure for testing the device at system level performed from a System Level Test point of view.

The BIST phase · the tester accesses to the instruction register of the TAP controller via the JTAG interface, activates the protocol for launching the Logic/Memory BISTs and downloads the results of the BIST for further analysis.

1.5 Software-based self-test

In software-based self-testing the test generation, test application and test response capturing are all tasks performed by embedded software routines that are

executed by the embedded processor itself, instead of being assigned to specially synthesized hardware modules as in hardware-based self-testing (BIST). Processors can, therefore, be re-used as an existing testing infrastructure for manufacturing testing and periodic/on-line testing in the field. Software-ased self-testing is a "natural", non-intrusive self-testing solution where the processor itself controls the flow of test data in its interior in such a way to detect its faults and no additional hardware is necessary for self-testing[24].

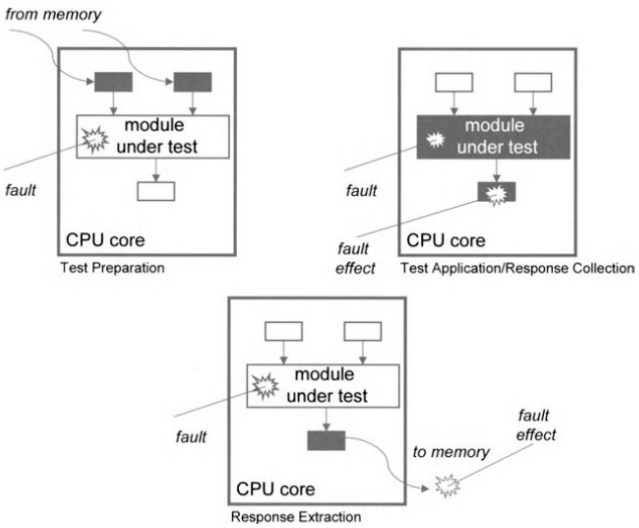

Figure 1.16: The three phases of SBST[8]

In software-based self-testing, the embedded processor executes a dedicated software routine or collections of routines that generate a sequence of test patterns according to a specific rationale. Subsequently, the processor applies each of the test patterns of the sequence to the component under test, collects the responses and finally stores them either in an unrolled fashion (each response is stored in memory) or in a compacted form (one or more test signatures). In a multi-processor SoC design, each of the embedded processors can test itself by software routines and then they can then apply software-based self-testing to the remaining cores of the SoC.

Self-test routines are stored in instruction memory (IRAM), in a process called *cache-resident testing*[15] and the data they need for execution are stored in data memory (DRAM), as depicted in Figure 1.16. Both transfers (instructions and data) are performed using external test equipment (ATE), a microcontroller in the context of System-Level-Test. Tests are applied to components of the processor core (CPU core) during the execution of the self-test programs and test responses are stored back in the data memory.

The application of software-based self-testing to a processor core manufacturing testing consists of the following steps:

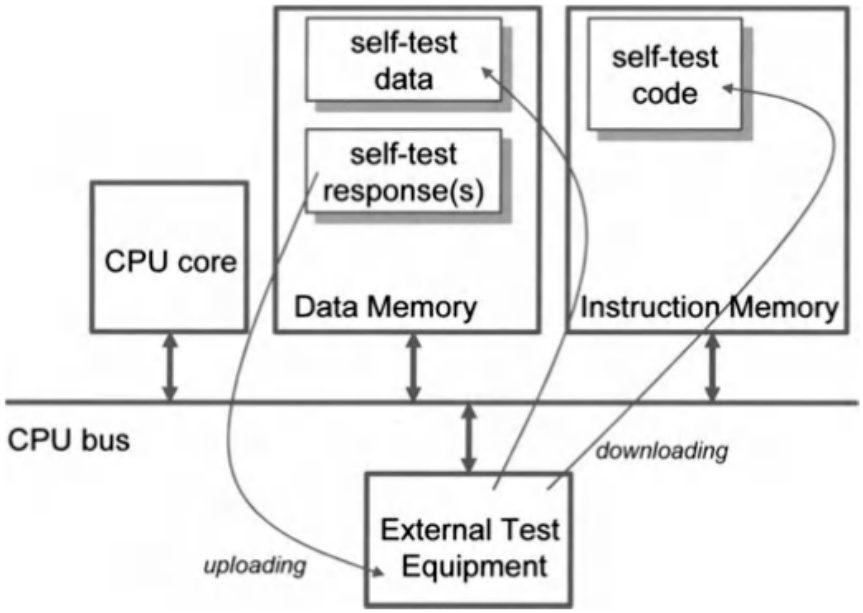

Figure 1.17: Software-based self-testing for a microprocessor[8]

- The self-test code is downloaded to the embedded instruction memory of the processor via the external test equipment, which has access to the internal bus.
- The embedded code will perform the self-testing of the processor. Alternatively, the self-test code may be ”built-in” in the sense that it is permanently stored in the chip in a ROM or flash memory. In this case, there is no need for a downloading process and the self-test code can be used many times for periodic/on-line testing of the processor in the field.
- The self-test data is downloaded to the embedded data memory of the processor via the same external equipment. Self-test data may consist of:

  - Parameters, variables and constants of the embedded code

  - Test patterns that will be explicitly applied to internal processor modules

  - The expected fault-free test responses (golden signature) to be compared with actual test responses. Downloading of self-test data does not exist if on-line testing is applied and the self-test program is permanently stored onboard.
- Control is transferred to self-test program candidate which starts executing. Test patterns are applied to internal processor components via the processor instructions to detect their faults. Components' responses are collected in registers and/or data memory locations. Responses may be collected in an unrolled manner in the data memory or may be compacted using any known test

response compaction algorithm, or instruction themselves. In the former case, more data memory is required and test application time may be longer, but, on the other hand, aliasing problems are avoided. In the latter case, data memory requirements are smaller because only one, or just a few, self-test signatures are collected, but aliasing problems may appear due to compaction.

- After self-test code completes execution, the test responses previously collected in data memory, either as individual responses for each test pattern or as compacted signatures are transferred to the external test equipment far evaluation.

Chapter 2

Case Study

SoC manufacturing is subject to the entirety of yield reduction phenomena described in Section 1.1, and one of the most challenging of those are faults in block-/block or die/die interconnects. Both interconnection between architectural blocks, such as the Arithmetic Logic Unit to the Register file, and those bridging a core die to a peripheral die – so-called ”chiplet” architecture.

2.1 The ”Bernina” SoC

The System-on-Chip under examination is manufactured by STMicroelectronics, it is nicknamed ”Bernina”; it is actually a SPC58NN84x microprocessor, of the SPC5 family of Automotive Microcontrollers, based on the PowerPC architecture, designed for high reliability applications. The key specifications include:

Core count Six e200z4256 cores, three user-controllable and two checker cores

Operating Frequency 200 MHz frequency ceiling via dual PLL

Instruction set Variable Length Encoding support, 16-bit versions of some instructions to reduce code footprint

Caches 128 KB on-chip general-purpose SRAM | 384 KB data RAM per core

Temperature −40 °C to 165 °C operating range

Longevity 15 years guaranteed

The process for high-volume SoC manufacturing[15] is shown in Figure 2.1:

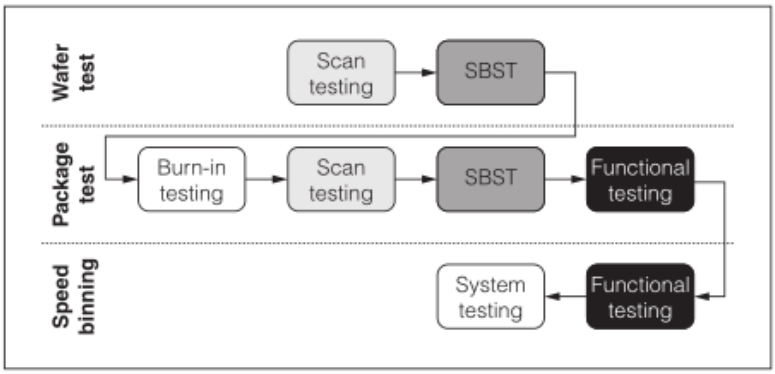

Figure 2.1: Typical high-volume manufacturing test flow

2.2 System Level Test framework

System Level Test attempts to emulate the operating environment of the Device-Under-Test, for a variety of purposes described in Section 1.4. In order to more effectively reduce the added costs deriving from the inclusion of this procedure in the test plan, the System-Level-Test board proposed is a modified version of the Burn-In board, shown in Figure 2.2.

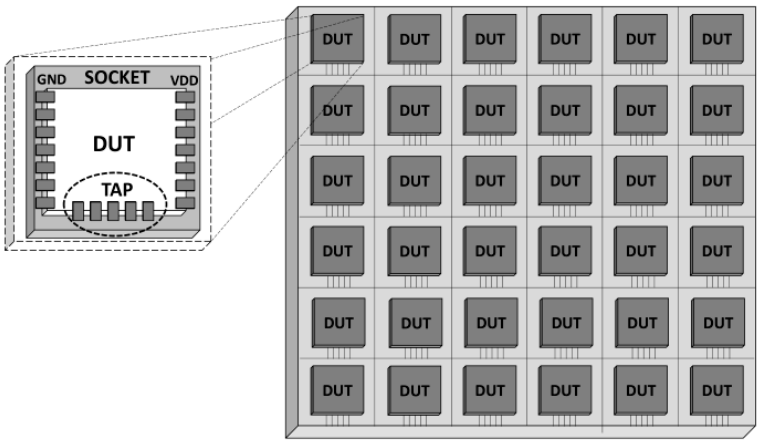

Figure 2.2: An industry standard Burn-In board [19]

System Level Test often complements the other steps of a test flow, which include Wafer Sort, Burn-In and Final Test, using functional test [19]. The functional test complements the structural test because it covers some defects that structural test does not detect. For example, the functional test works at the system operational speed while some Design for Testability techniques do not. Moreover, System Level Test exercises the system exactly in the same conditions of the operational phase[12]. System Level Test is sometimes used as an effective method to lower the

defectivity, often measured in terms of Defective Parts Per Million (DPPM). System Level Test increases the quality of the shipped products, which is crucial in safety-critical applications. System Level Test addresses both defects and marginalities [2]. In our view, a possible defect (physical) is always present, and test conditions may only change the manifestation of such. Conversely, a marginality may not be present in a subset of possible test conditions, and may impact the functionality on specific PVT (Process, Voltage, and Temperature) condition(s), only. A marginality might also be active only after a specific functional sequence (including software) is applied. Detection of marginalities have always been delegated to bench-top validation, under the assumption that a reduced number of corner cases are enough to view all marginalities symptoms.

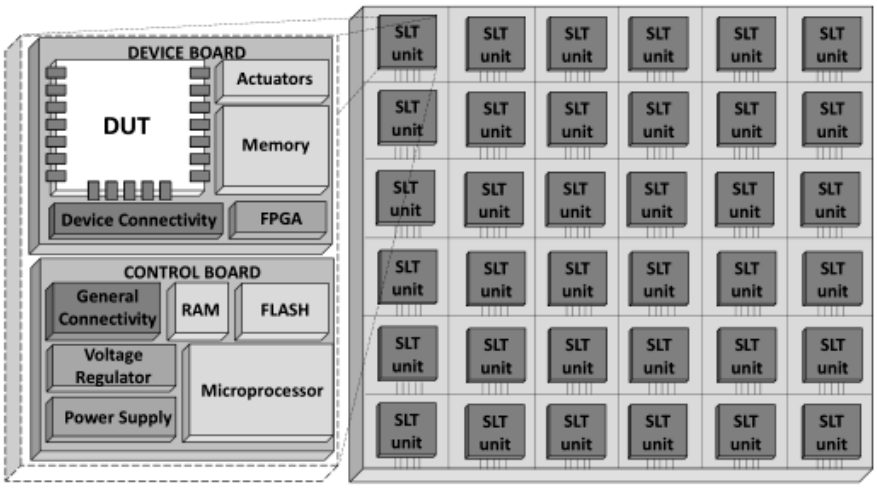

Figure 2.3: A "macroboard" with detail of a DUT daughterboard [19]

2.2.1 Supply voltage manipulation

STMicroelectronics proposes augmenting the Burn-In board, with the aim of phasing out expensive ATEs, typically operating on one device at once, with a System-Level-Test approach. In the early phases, with relatively high defectivity and consequent long time for the screening of early-life fails, the adoption of a flow with both Burn-In and System Level Test is economically affordable. In the proposed approach, Burn In and System Level Test duration are combined, so that the test procedure is shorter, thus more economically convenient[12].

The SLTBI approach also removes the extra time required for the load/unload operations. For this purpose, each device is placed in a SLT board, shown in Figure 2.3. This approach allows for parallel testing of multiple samples, and fine tuning the supply voltage of each device. Moreover, the device's Test Access Points can be

exploited by the onboard microcontroller to apply test patterns, provided they fit on the memory of the System Level Test board.

In this case study, once the best candidate programs generated by the evolutionary algorithm are selected, they are meant to be tested at varying supply voltage. For this purpose, the core voltage rail ($V_{nom} = 1.25 \pm 0.1\,\mathrm{V}$) is fed by the onboard regulator. The test plan can follow one of two paradigms:

Overvoltage · The DUT undergoes a "derating-like" process, running with a supply exceeding the manufacturer specification, in order to maximize the heat produced by the device and excite latent defects.

Undervoltage · The voltage rail is set to below specification, with the aim of causing what's known as *voltage droop*, shown in Figure 2.4. The blue line shows the absolute minimum rating, red the hypothesised level where delay faults would manifest.

One might object that the voltage supply could be reduced to the supposed critical level in the first place and more reliably produce the test conditions, instead of relying on dynamic droop phenomena. However, the SoC in question includes an automatic reset mechanism when the input voltage falls below 1.1 V, and would not allow for program execution. Moreover, dynamic phenomena are the most critical when in comes to at-speed testing, hence the choice performed when it comes to undervoltage.

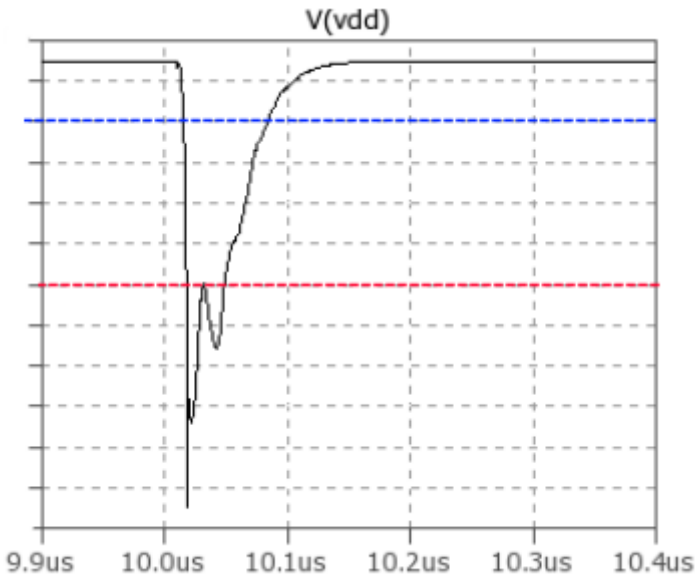

Figure 2.4: A simulation of voltage droop[10].

2.3 Instruction pool

As mentioned in Section 1.3.1, the evolutionary algorithm evolves an Assembly test program, generating both instruction and operands for the core to execute. The aim of the selecte test program is to maximize the Toggle Activity, which Section 1.1.3 shows is directly proportional to dynamic power consumption. A sudden rush in dynamic power, that is a spike in current consumption, is expected to create a voltage droop at the target module.

In order to focus the evolution on the switching of a given module, the Instruction Set Architecture of the SoC in question was reduced to a set of instruction relevant to the module in question. The procedure was performed on two pair of logic entities, each with the instructions listed in Table 2.5.

1. **Integer Adder + Register File**

2. **Logic Unit + Register File**

Adder		**Logic**	
Type	**Mnemonic**	**Type**	**Mnemonic**

Short, 1 operand	`se_neg`	Short, 1 operand	`se_not`
			`se_extsb`
			`se_extsh`
			`se_extzb`
			`se_extzh`

Short, 2 operands	`se_add`
	`se_add`
	`se_subf`
	`se_subi`
		Short, 2 operands	`se_and`
			`se_and.`
			`se_or`
			`se_andc`
			`se_andi`

Adder

Type	Mnemonic
2 operands	addme
	addmeo.
	addze
	addzeo.
	neg
	nego.
	subfme
	subfmeo.
	subfzeo.
3 operands	add
	add.
	addc
	addco.
	adde
	addeo.
	subf
	subfo.
	subfco.
	subfe
	subfe.
	subfeo
	subfeo.
	e_add16i
	e_addi
	e_addi.
	e_addic
	e_addic.
	e_subfic.

Logic

Type	Mnemonic
2 operands	e_andi
	e_andi.
	e_ori
	e_ori.
	e_xori
	e_xori.
	e_and2i.
	e_and2is.
	e_or2i
	e_or2is
	cntlzw
	cntlzw.
3 operands	andc
	andc.
	and
	and.
	eqv
	eqv.
	nand
	nand.
	nor
	nor.
	or
	or.
	orc
	orc.
	xor
	xor.

Figure 2.5: Instructions pools used for candidate evolutions.

2.4 Graph-based program classificator

Simply put, when a pair of Assembly instruction encounter this kind of data hazard, the latter overwrites the content of its destination register, which had also been used as destination by the former. Effectively, this event masks the first, and therefore prevents detection of a possibly occurred fault (*masking*). In order to prevent this occurrence but still mantain a reasonable program body size, Read-After-Write operation must be performed on any instruction whose destination register is overwritten. The state of the data registers at the end of execution is then compared to what's known as *golden output*, produced by simulation. If the two differ, a fault was excited and it produced an error.

To accomplish this task, an algorithm was put in place, classifying programs whose instructions were completely masked in a graph-like structure.

As mentioned in Section 2.3, each individual is ranked using a fitness function. This function evaluates the switching activity in the target module produced by the candidate program and the flow of the instructions.

A fault can occur at any point during the program execution, and therefore the output of each instruction should be observable at the end of the execution, creating a completely different test signature at the end. Specifically, let us take as example Figure 2.6

```
1 50_percent::          1  75_percent::          1  100_percent::
2 ADD  R6, R5, R3       2  ADD  R4, R5, R3       2  ADD  R6, R5, R3
3 ADDI R6, R4, 13       3  ADDI R6, R4, 13       3  ADDI R8, R6, 13
4 SUBI R7, R9, 12       4  SUBI R4, R9, 12       4  SUBI R7, R9, 12
5 SUB  R7, R2, R3       5  SUB  R7, R2, R3       5  SUB  R5, R2, R3
```

Figure 2.6: Program body observability examples

Listing 2.6 shows three pieces of code, each respectively ranked as 50%, 75 % and 100% observable by the proposed algorithm. Specifically, the leftmost piece of code shares the destination register between two instructions, without previously taking as input the previous result.

The center 75% program body instead performs what's known as Read-after-Write, that is instruction 1 produces a result in R4, then instruction 2 uses R4 as input operand. Effectively, this operation *hashes* instruction 1 with instruction 2, and stores the hash in R6. R4 can now be safely overwritten, and R6 can then be fed to another instruction and so on.

Lastly, the program with 100% coverage shows another approach using a different register for each instruction. As one can clearly see, this method is severely hampered by the limited amount of Register File entries available, and would not

be practical in a test case scenario.

Generalizing, the problem is easily modelled by a Directed Acyclic Graph (DAG), the structure of which was outlined in Section 1.3.2. The problem of estimating the *data dependency* metric to feed back to μGP can then be solved by means of a *graph colouring* algorithm.

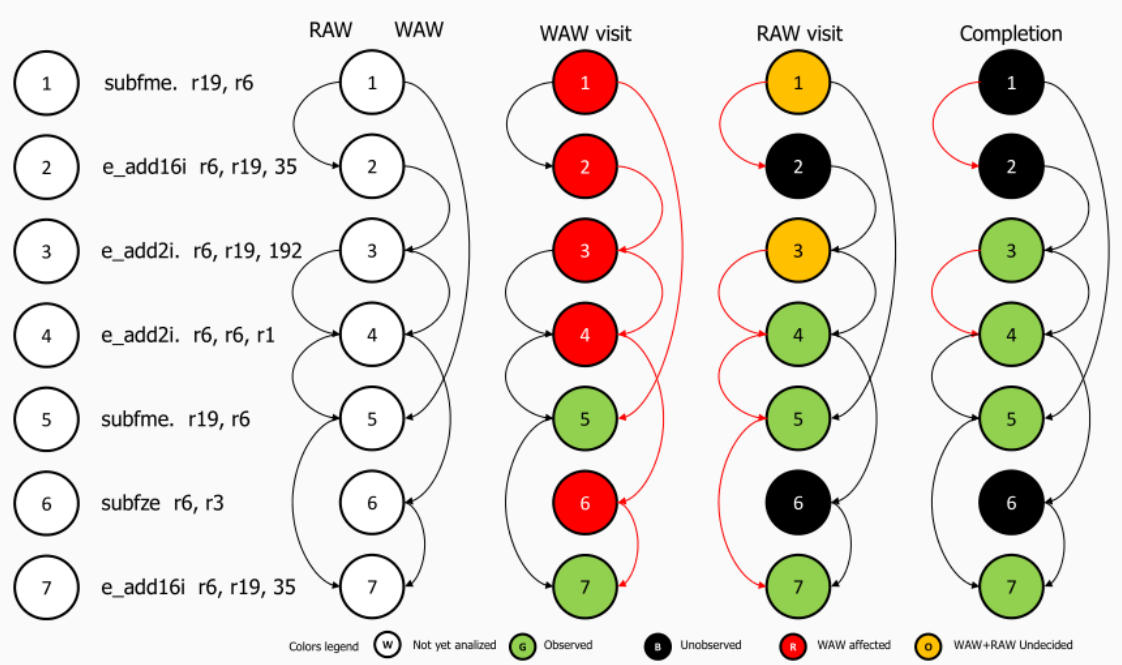

Figure 2.7: A sample of instruction graph representation

The dependency evaluation algorithm follows a rather simple approach, that is:

1. The Assembly program's body is cropped, and its contents wrote to a temporary location.

2. The program body is parsed, and each instruction is translated to a node, with its Read-After-Write and Write-After-Read dependencies determining its edges.

3. A **Begin** and **End** node are appended to the head and tail of the graph, and connected appropriately.

4. The program starts navigating the graph from the **End** node, identifying WAW hazards.

5. The program performs a second pass, identifying RAW hazards. At this point, some instruction's state may still be undefined.

6. A last pass resolves the undecided node, depending on whether they are connected to a masked or observable node.

7. The coloured nodes are counted, and the dependency percentage can now be produced.

For a 200-instruction program body, the evaluation takes approximately 5 seconds, which is definitely negligible with respect to the average candidate simulation time of about 7 minutes.

2.4.1 Switching activity

In the framework of maximizing the power draw, or better yet maximizing the toggle count of each and every signal inside the target module, the simulator NCsim was again employed to simulate the execution of every program. The structure of each test program is the following:

```
1  prologue::
2
3  // ENABLE ICACHE
4
5  // CONFIGURE PLLs
6
7  loop::
8
9  // PROGRAM BODY
10 // looping 100 times
11
12 epilogue::
```

The simulator records the activity of the whole core from labels `loop` to `end`, and a generates a Toggle Count Format file (*.tcf) showing the toggle history of each net in the scope.

2.4.2 Fitness evaluation

The applicability of evolution-based test program selection entirely relies on the validity of the fitness function employed. This section will evaluate the steps performed in creating such function and the reason behind this choice.

The evolution is aimed at maximizing the toggle activity of an arbitrary entity in a core, in a framework of limiting the power reaching the module and therefore possibly exciting a dormant delay fault. The nature of such a fault is high volatility, therefore it is essential that one stores and is able to observe the result of each test operation. For this purpose, a dependency evaluation algorithm had to be employed, described in detail in Section 2.4.

The toggles belonging to the target module are accumulated and normalized with respect to the execution time, in order to discourage the creation of long

program bodies. Finally, a collection of fitness parameters was determined, and its final structure is the following:

$$\texttt{<dependency>} < \frac{\texttt{Toggles}}{\texttt{Execution time}} < \frac{\texttt{1}}{\texttt{Variance}} >$$

Both mean value and variance were computed after parsing the simulation results, using Python 3 library `numpy`.

2.5 Evolution results

2.5.1 Integer Adder

The first optimization process ran was aimed at the Integer Adder Module, since it was the smaller of the two. The statistics regarding this evolution are:

	Individuals	**Generations**	**Evolution time**
Adder	2641	97	17d 6h

The results were obtained parsing the fitness files of all the individuals of the generation, then plotted using MATLAB. The primary fitness graph is depicted in Figure 2.8:

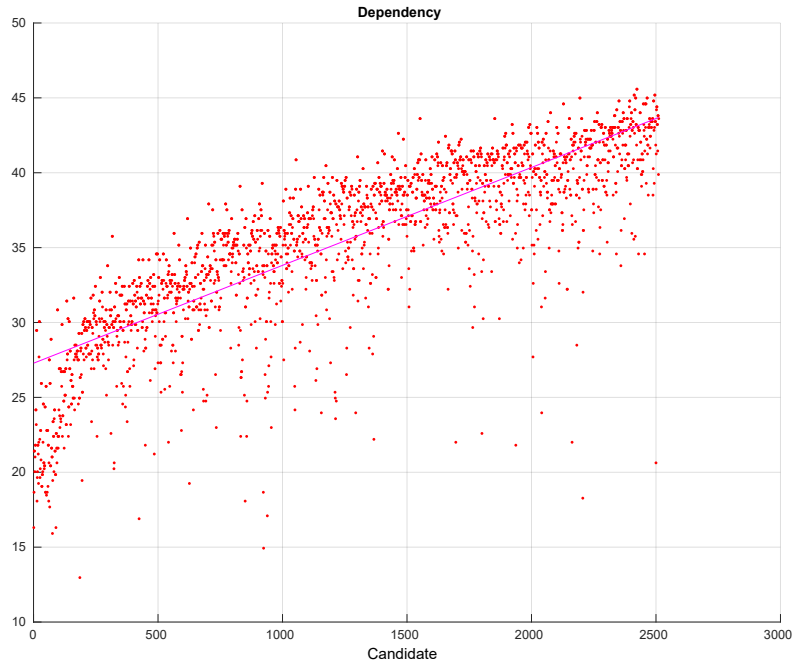

Figure 2.8: Primary fitness statistics for the Integer Adder run

The assumptions made in Section 1.3.1 are confirmed by the graph, the instruction interdependency increases steadily over time, the tendency of which is shown by the magenta line. The population diversity is also increased, manifesting in a ”widening” of the primary fitness values over time. When it comes to the secondary and tertiary fitnesses, their graphs are depicted in Figure 2.9.

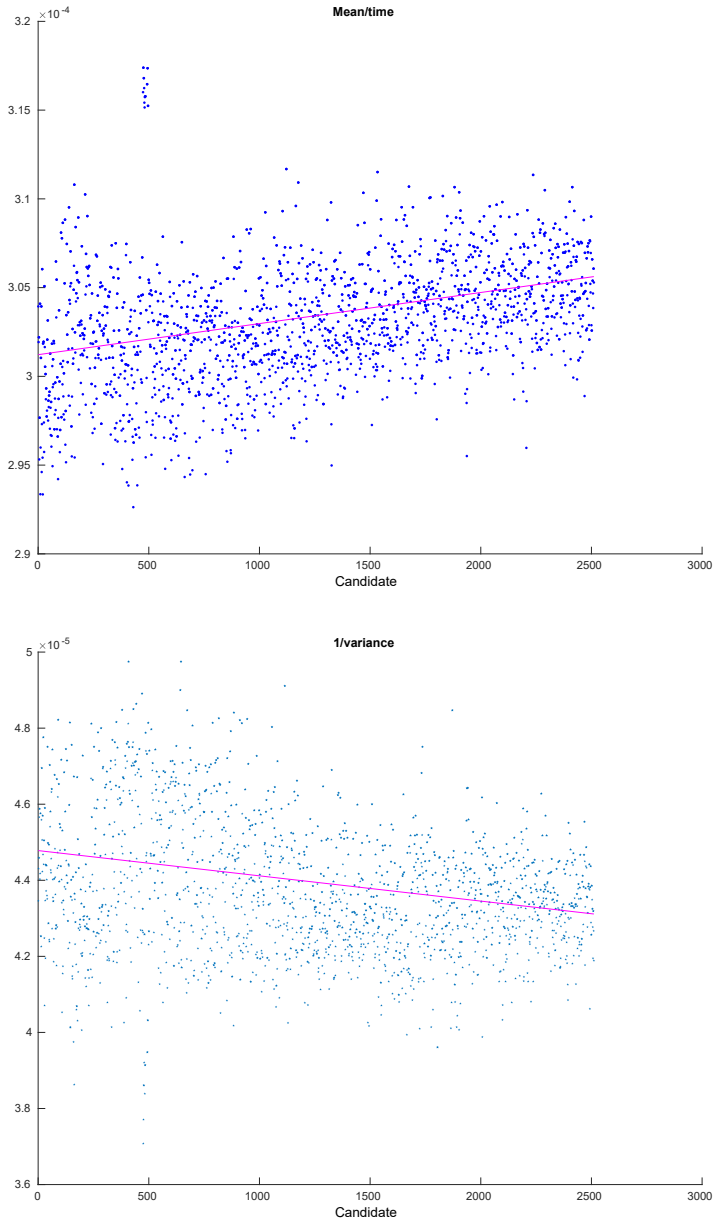

Figure 2.9: Secondary and tertiary fitness statistics for the Integer Adder run

The secondary fitness exhibits a behaviour similar to the primary, progressively increasing over time, which matches the assumptions made in the preliminary steps. When it comes to the inverse of the variance, which was expected to grow as well, it did not. The reason for this deviation from the expectations is unclear: the behaviour of this parameter is identical in the Logic Unit evolution, therefore it could be theorized that an increase of toggles' mean value and a simultaneous decrease in variance is impossible.

Once the statistics of the evolutionary run were imported in MATLAB, they were used to select the best individuals for each fitness statistic, shown in Figure 2.10.

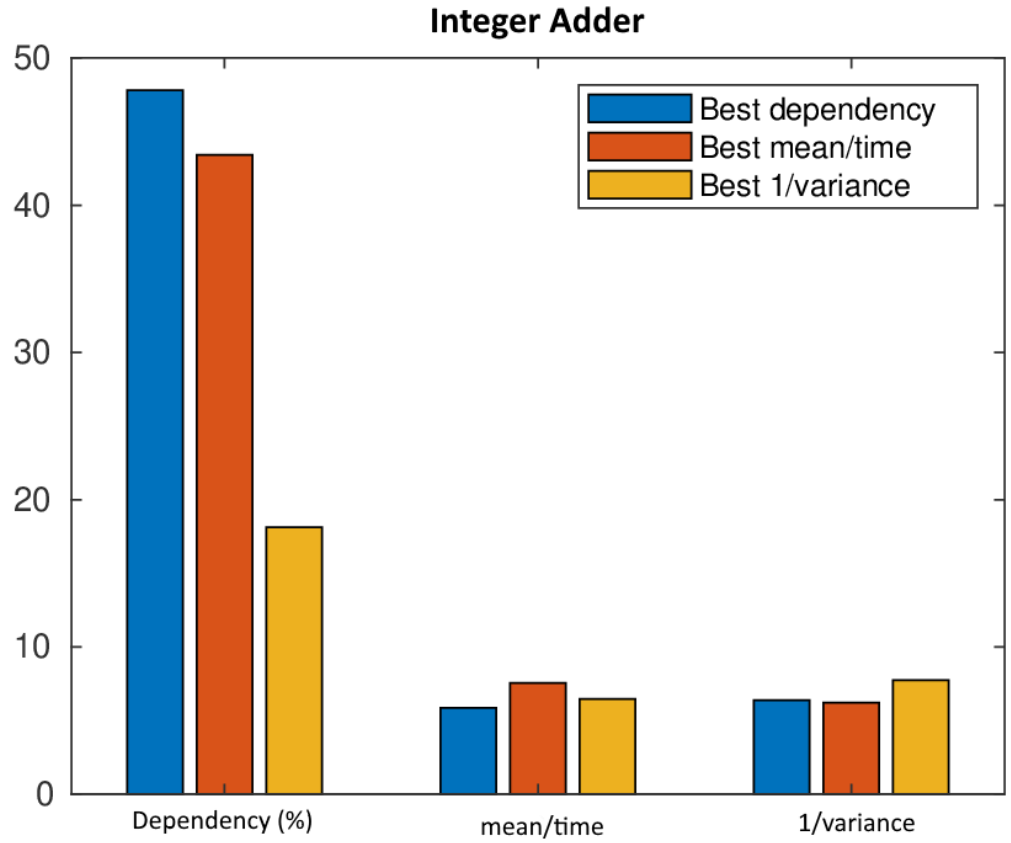

Figure 2.10: Best Integer Adder individuals with respect to each fitness metric

As one can see, the difference between the statistics of each individual is relatively small, due to the comparatively short evolution duration.

2.5.2 Logic Unit

The Logic Unit is the larger of the two units under examination, and for this reason its evolution was allowed to continue for a much longer duration, even though it did not reach any of the stopping conditions discussed in Section 1.3.1. The statistics relative to this evolutionary run are the following:

	Individuals	**Generations**	**Evolution time**
Logic Unit	23929	1523	83d 22h

The primary fitness graph of this evolutionary run is depicted in Figure 2.11.

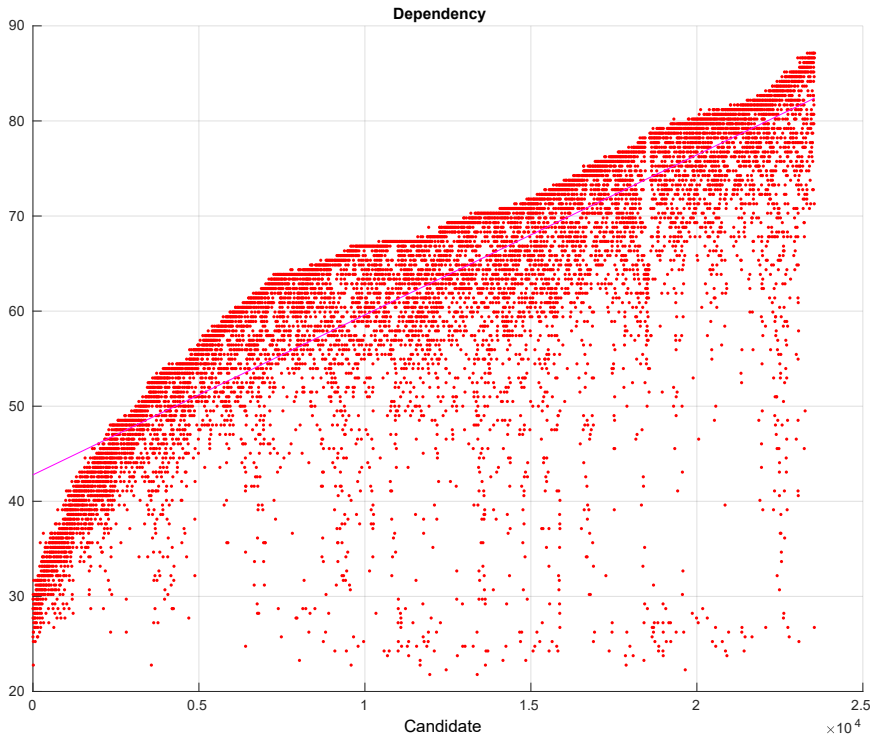

Figure 2.11: Primary fitness statistic for the Logic Unit run

As was the case for the Integer Adder evolution, the primary fitness organically grows maintaining population diversity. To better visualize the primary fitness behaviour, the statistics were filtered to only show the best individual for each generation, in Figure 2.12.

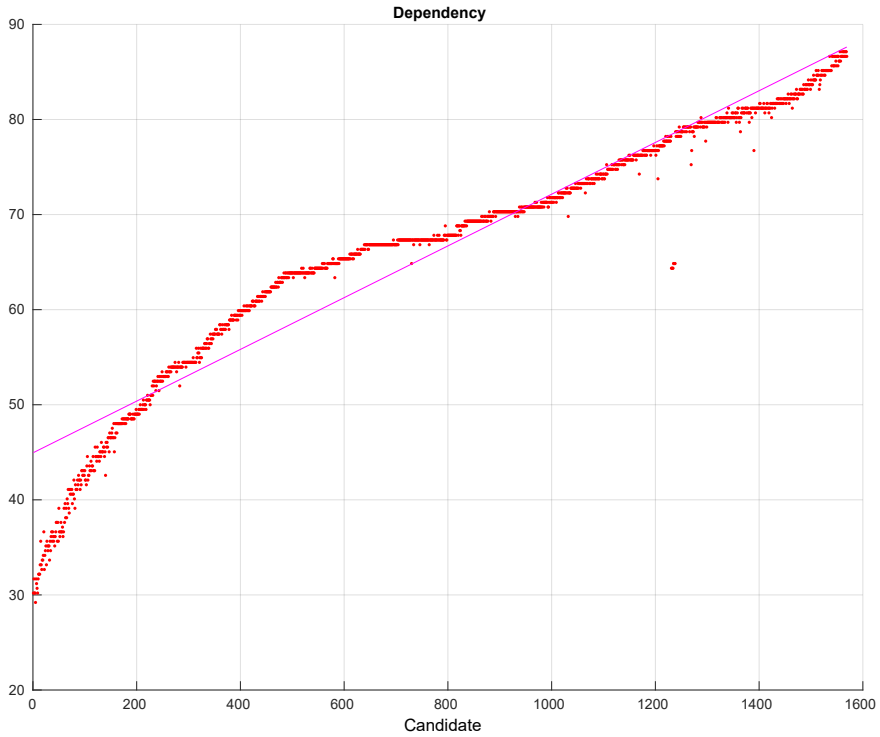

Figure 2.12: Best individuals w/ respect to the Logic Unit primary fitness

The evolutionary run was determined to be complete, since the primary fitness exceeded 85%. The remaining masked instructions could be easily modified to be observable without excessively compromising the switching activity produced by the program. When it comes to secondary and tertiary fitnesses, their graphs are depicted in Figure 2.13.

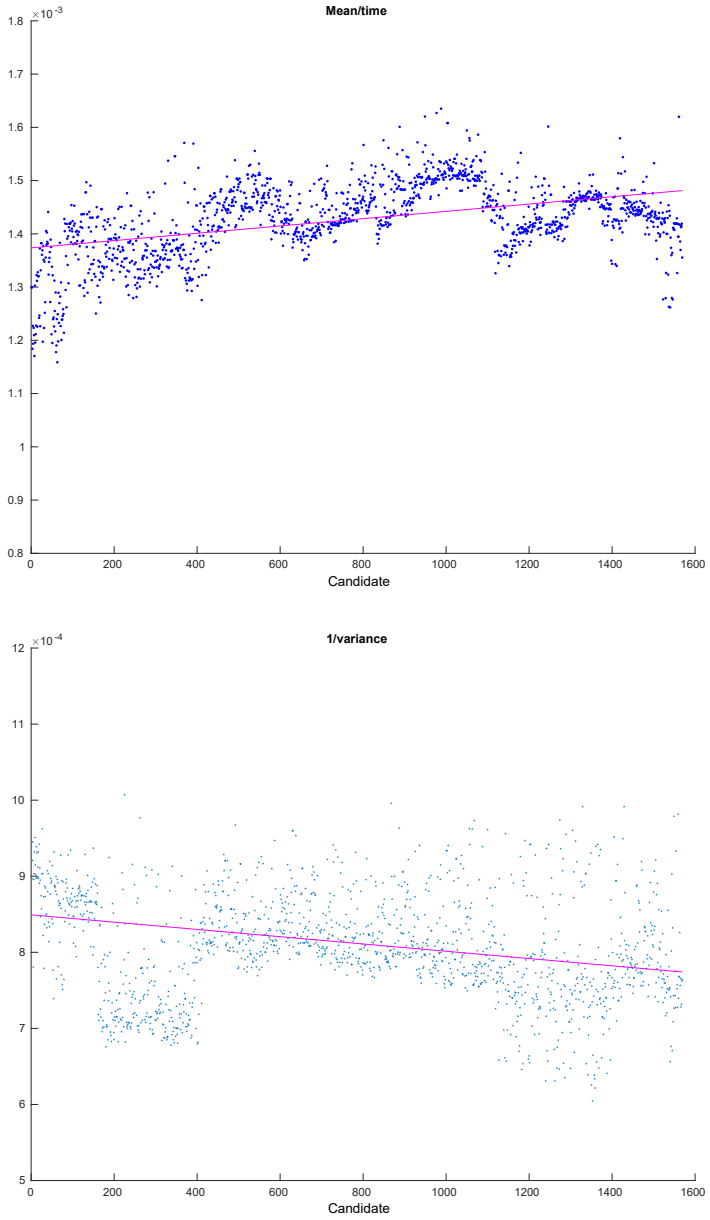

Figure 2.13: Best candidates w/ respect to Logic Unit secondary and tertiary fitness

The ratio between the mean toggles value and the program body duration again oscillates but largely grows over time, whereas the inverse of the variance does not improve over time. The reason behind this failure in converging to the expected behaviour is not known at the present time, but it is possible that improvements in both mean value and variance can not be simultaneously achieved.

The best three individuals were again selected and their fitnesses plotted in the bar graph in Figure 2.14.

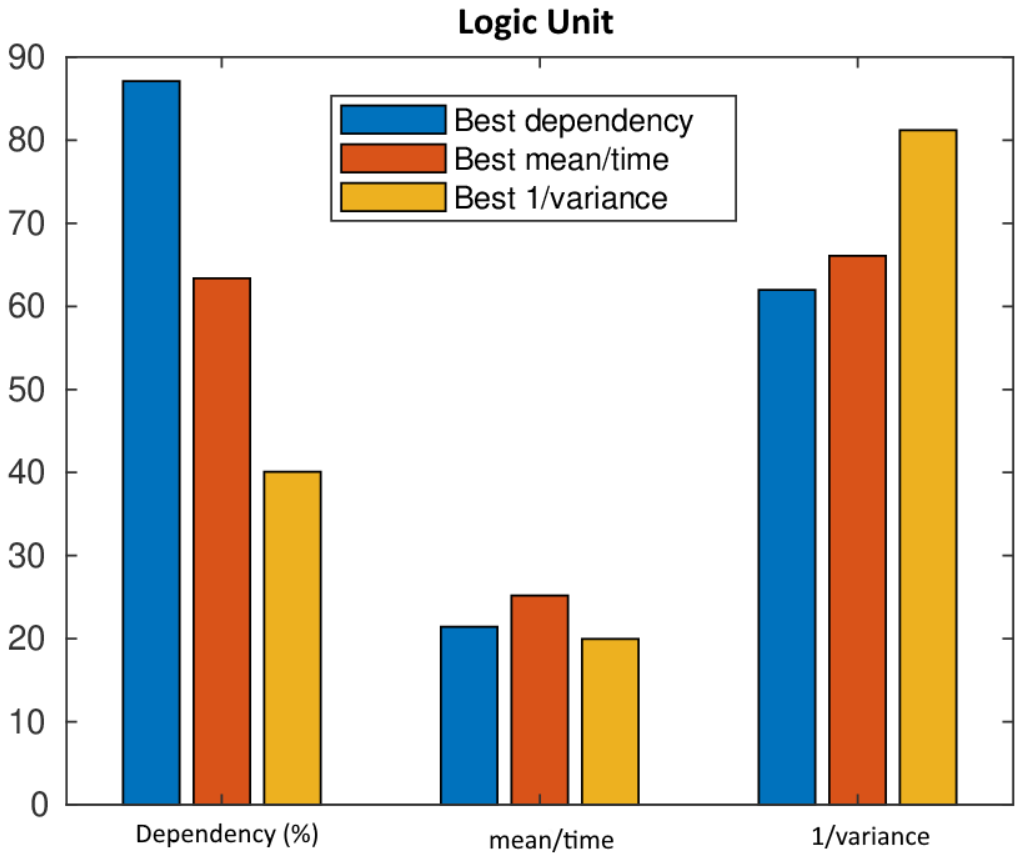

Figure 2.14: Best Logic Unit individuals with respect to each fitness metric

The results relative to the Logic Unit look more promising than those of the Integer Adder, specifically when it comes to the average toggles value, while on the other hand their variance is significantly greater. It's possible the evolutionary run would have optimized the secondary fitnesses with more time available, or with increased processing power dedicated to simulation.

Regardless, once the most capable individuals were obtained, their performance was evaluated on physical chips.

Chapter 3

Physical testing

The present chapter discusses the steps performed in verifying the correct functioning of the test programs on the real System-on-Chip device. Moreover, this chapter will describe the hardware and software employed to accomplish this task.

3.1 Hardware

In order to program and manage the Device-Under-Test, a host platform was required. The choice fell on STMicroelectronics' STeaLTh shown in Figure 3.2, which features an integrated digitally controlled regulator to alter the DUT's core voltage rail. The general schematic of the test environment is shown in Figure 3.1

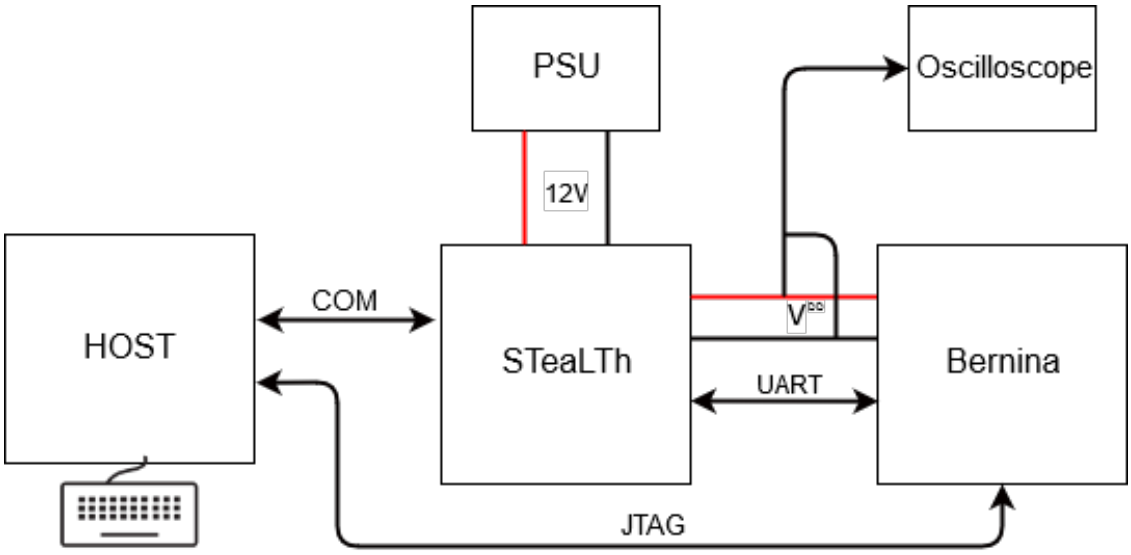

Figure 3.1: A general schematic of the physical testing setup

This board communicates both with the host terminal via a COM port and to the DUT via UART. The DUT is placed in a socket on another board, depicted in Figure 3.3, which integrates the voltage regulators for the 5 V, 3.3 V and 1.25 V rails, the latter of which is also referred to as V^{DD}. Each voltage rail can be manually disabled when needed, and this operation was performed when performing voltage manipulation tests, discussed in Section 3.3. The voltage regulator on the STeaLTh board is fed by a bench PSU by means of banana plugs. Moreover, an oscilloscope is set to monitor the voltage being produced by the STeaLTh board and fed to Bernina, in order to monitor its true value, behaviour and ripple.

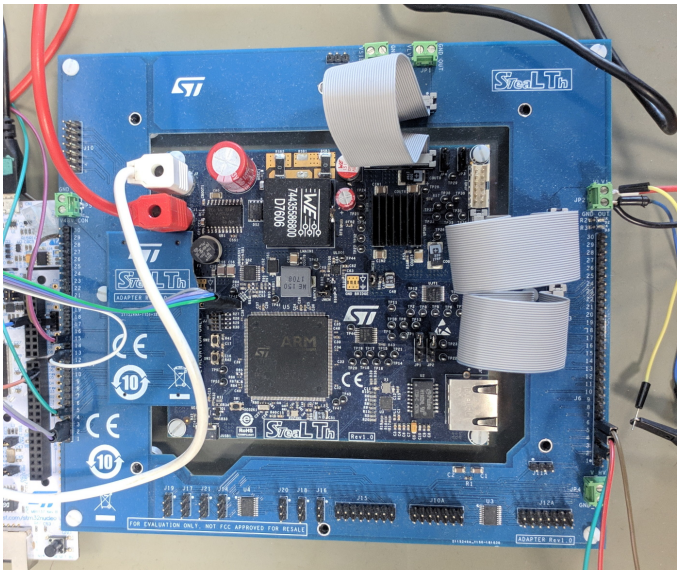

Figure 3.2: The STeaLTh board

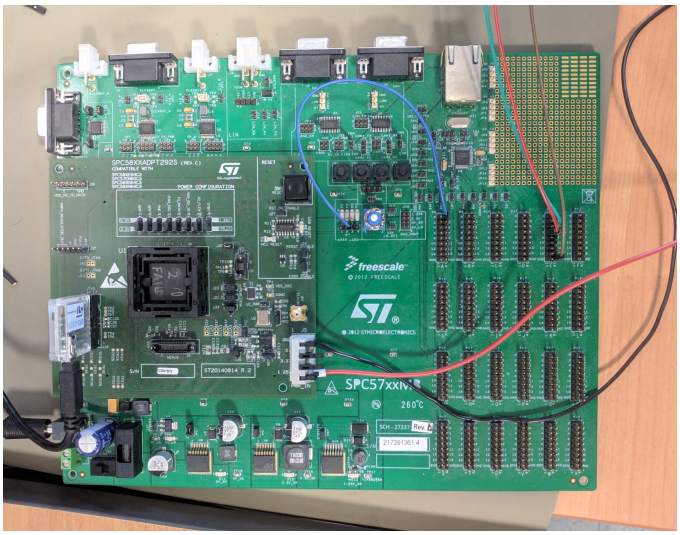

Figure 3.3: The Bernina board

3.2 Software

The software running on both Bernina and STeaLTh boards is based on FreeRTOS. The reason for this choice instead of a from-scratch implementation is the availability of software libraries on this platform. Specifically, the UART communication between the DUT and the STeaLTh board , and from the STeaLTh board to the host PC would have been very involved if implemented from scratch.

The evolved test programs were written in Assembly, and the firmware in C language. This required that the stack frame created and destroyed by those programs was EABI compliant, that is not overwrite the stack created by routines calling it. Moreover, the test output must be returned to the C firmware for validation, therefore it was mandatory that it was placed in a specific register (R3) for the value to be correctly parsed.

The DUT is programmed by means of a JTAG interface directly from the host PC, and the software is designed to idle until it receives a specific message from the STeaLTh board . The test procedure is the following:

1. The DUT and STeaLTh board are programmed with their respective firmwares.
2. The STeaLTh board boots and enables its voltage regulator.
3. STeaLTh waits for a command from the host PC, specifying which test routine to run.
4. A valid command is received by the STeaLTh board : it optionally adjusts V_{DD} and forwards it to the DUT.
5. The DUT executes the specified test, whose length is approximately 10 minutes.
6. The DUT sends the test result back to the STeaLTh board .
7. Depending on the test routine being ran, the following steps may be performed:

   - V_{DD} is adjusted.

   - The DUT repeats the test more than once, each time reporting the result to the host.

In order to more effectively test the generated programs, several test routines were devised, picking from the following:

`ADDER` · The program with the greatest primary fitness produced by the Integer Adder run.

`LOGIC` · The program with the greatest primary fitness produced by the Logic Unit run.

`MEMTEST` · A March 7 algorithm testing the Data Memory of a given core.

The test routines employed are depicted in Table 3.1.

Test routine	**Core 0**	**Core 1**	**Core 2**
TEST1	LOGIC	LOGIC	LOGIC
TEST2	ADDER	ADDER	ADDER
TEST3	MEMTEST	LOGIC	MEMTEST
TEST4	MEMTEST	ADDER	MEMTEST

Table 3.1: The test routines employed in the present case study.

3.3 Experimental results

Employing the procedures described in Section 3.2, the tests were carried out on a number of devices. STMicroelectronics provided three bins of System-on-Chips to perform experiments on. The test procedures executed by ST reported the bins as:

- **BIN A** - Faulty
- **BIN B** - Faulty
- **BIN C** - Good

The manufacturer provided no additional information about the faulty chips other than that they would fail at different supply voltages. Supposedly, this type of defect would render the samples more prone to failing using the procedures for supply voltage manipulation described in Section 3.1.

The test procedures of Table 3.1 were performed a variety of times and at different supply voltage levels. The results are reported in Table 3.2.

Bin	Device	Test routine	Supply voltage	Test time (mins.)	Result
BIN A	6	TEST1	1.092 V	10	*Pass*
BIN A	6	TEST1	1.097 V	25	*Pass*
BIN A	6	TEST1	1.391 V	10	*Pass*
BIN A	6	TEST1	1.395 V	25	*Pass*
BIN A	6	TEST2	1.091 V	10	*Pass*
BIN A	6	TEST2	1.092 V	25	*Pass*
BIN A	6	TEST2	1.393 V	10	*Pass*
BIN A	6	TEST2	1.394 V	25	*Pass*
BIN A	6	TEST3	1.091 V	10	*Pass*
BIN A	6	TEST3	1.092 V	25	*Pass*
BIN A	6	TEST3	1.393 V	10	*Pass*
BIN A	6	TEST3	1.394 V	25	*Pass*
BIN A	6	TEST4	1.096 V	10	*Pass*
BIN A	6	TEST4	1.092 V	25	*Pass*
BIN A	6	TEST4	1.393 V	10	*Pass*
BIN A	6	TEST4	1.393 V	25	*Pass*
BIN B	3	TEST1	1.092 V	10	*Pass*
BIN B	3	TEST1	1.094 V	25	*Pass*
BIN B	3	TEST1	1.392 V	10	*Pass*
BIN B	3	TEST1	1.397 V	25	*Pass*
BIN B	3	TEST2	1.097 V	10	*Pass*
BIN B	3	TEST2	1.096 V	25	*Pass*
BIN B	3	TEST2	1.396 V	10	*Pass*
BIN B	3	TEST2	1.391 V	25	*Pass*
BIN B	3	TEST3	1.091 V	10	*Pass*
BIN B	3	TEST3	1.092 V	25	*Pass*
BIN B	3	TEST3	1.393 V	10	*Pass*
BIN B	3	TEST3	1.397 V	25	*Pass*
BIN B	3	TEST4	1.092 V	10	*Pass*
BIN B	3	TEST4	1.094 V	25	*Pass*
BIN B	3	TEST4	1.391 V	10	*Pass*
BIN B	3	TEST4	1.393 V	25	*Pass*

Table 3.2: Results of the candidate programs physical tests

As per Table 3.2, the devised test routines were performed both over and below the specified supply voltage. Despite extensive testing for both short and long durations to raise the chip temperature as much as possible, no failures were detected.

Moreover, the voltage regulator embedded on the STeaLTh board was verified to be extremely accurate and did not exhibit any kind of nonideality such as overshoot or voltage droop.

Chapter 4

Conclusions

The present thesis outlines a novel approach allowing for not only automatic test program generation, but also a hardware flow to test those programs at different supply voltages. The generation is based on an evolutionary algorithm to produce the set of assembly programs With the advent of even more higher performing devices, traditional functional approaches are not sufficient to guarantee the correct functioning of the devices. This methodology exploits μGP, an evolutionary algorithm to optimize and devise automatically programs written in Assembly language. The need to use an automatic test program generator rises from the necessity to cope with the increasing hardware component's complexity. Actually, the advances in microelectronics technologies allowed semiconductor manufacturers to deliver chips with ever-shrinking form fac- tors and ever-increasing switching frequencies [3]. This requires even more sophisticated verification mechanisms to best exploit the device's features and detect latent defects, since one can't rely on handwritten test programs anumore.

The use of an automatic method was demonstrated to dramatically help designers and engineers: Instead of checking massive random simulations looking for deviations from the golden device, validation experts can let the automatic test case generator work for the as long as it takes, and even examine and improve on the test programs it produces. Results such as those of the Integer Adder evolutionary run show that some intervention is required to best exploit the programs produced by μGP. Of course these steps demand expert and skilled engineers whose work is to carefully monitor simulations and understand which parts of the target modules are switching the least and why.

4.1 Future work

The test programs produced by the evolutionary algorithm were not capable of detecting any of the latent faults known to be present in the devices under test.

This results does not invalidate the premise of making use of μGP to produce test programs. The proposed approach should be targeted at larger modules in the System-on-Chip under test, such as the Floating Point Unit or the Control Unit. Such large entities would surely draw a greater amount of current, and therefore exacerbate greatly the phenomenon of voltage droop.

Moreover, if one had the computing capabilities, could let the evolutionary algorithm determine which module or collection of modules to target.

Bibliography

[1] Brahme and Abraham. "Functional Testing of Microprocessors". In: *IEEE Transactions on Computers* C-33.6 (1984), pp. 475–485. DOI: 10.1109/tc.1984.1676471.

[2] Z. Conroy et al. "A practical perspective on reducing ASIC NTFs". In: *IEEE International Test Conference*. IEEE, 2005. DOI: 10.1109/test.2005.1583992.

[3] F. Corno, E. Sanchez, and G. Squillero. "Evolving Assembly Programs: How Games Help Microprocessor Validation". In: *IEEE Transactions on Evolutionary Computation* 9.6 (2005), pp. 695–706. DOI: 10.1109/tevc.2005.856207.

[4] F. Corno et al. "Code Generation for Functional Validation of Pipelined Microprocessors". In: *Journal of Electronic Testing* 20.3 (2004), pp. 269–278. DOI: 10.1023/b:jett.0000029460.80721.4d.

[5] S. Davidson. "Towards an Understanding of No Trouble Found Devices". In: *23rd IEEE VLSI Test Symposium (VTS'05)*. IEEE Comput. Soc, 2005. DOI: 10.1109/vts.2005.86.

[6] A. E. Eiben and E. Smith James. *Introduction to Evolutionary Computing*. Springer, 2003.

[7] J. R. English, Li Yan, and T. L. Landers. "A modified bathtub curve with latent failures". In: *Annual Reliability and Maintainability Symposium 1995 Proceedings*. IEEE, 1995. DOI: 10.1109/rams.1995.513249.

[8] Dimitris Gizopoulos, A. Paschalis, and Yervant Zorian. *Embedded Processor-Based Self-Test (Frontiers in Electronic Testing Book 28)*. Springer, 2013. ISBN: 978-1-4020-2801-4.

[9] George Harman. *Wire Bonding in Microelectronics*. McGraw-Hill Education Ltd, Mar. 1, 2010. ISBN: 0071476237. URL: https://www.ebook.de/de/product/7441970/george_harman_wire_bonding_in_microelectronics.html.

[10] Matthew Seetharam A. Holtz, Seetharam Narasimhan, and Swarup Bhunia. "On-die CMOS voltage droop detection and dynamiccompensation". In: *Proceedings of the 18th ACM Great Lakes symposium on VLSI - GLSVLSI '08*. ACM Press, 2008. DOI: 10.1145/1366110.1366122.

[11] M.S. Hsiao, E.M. Rudnick, and J.H. Patel. "Peak power estimation of VLSI circuits: new peak power measures". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8.4 (2000), pp. 435–439. DOI: 10.1109/92.863624.

[12] Artur Jutman, Matteo Sonza Reorda, and Hans-Joachim Wunderlich. "High Quality System Level Test and Diagnosis". In: *2014 IEEE 23rd Asian Test Symposium*. IEEE, 2014. DOI: 10.1109/ats.2014.62.

[13] Way Kuo. "Reliability Enhancement Through Optimal Burn-In". In: *IEEE Transactions on Reliability* R-33.2 (1984), pp. 145–156. DOI: 10.1109/tr.1984.5221760.

[14] Hermes Liu et al. "Advanced method to monitor design-process marginality for 65nm node and beyond". In: *2008 IEEE/SEMI Advanced Semiconductor Manufacturing Conference*. IEEE, 2008. DOI: 10.1109/asmc.2008.4528996.

[15] Mihalis Psarakis et al. "Microprocessor Software-Based Self-Testing". In: *IEEE Design & Test of Computers* 27.3 (2010), pp. 4–19. DOI: 10.1109/mdt.2010.5.

[16] G. J. Van Rootselaar and B. Vermeulen. "Silicon debug: scan chains alone are not enough". In: *International Test Conference. Proceedings (IEEE Cat. No.99CH37034)*. Int. Test. Conference, 1999. DOI: 10.1109/test.1999.805821.

[17] Annachiara Ruospo, Ernesto Sanchez Sanchez Edgar, and Matteo Sonza Reorda. "An Evolutionary Approach for Functional Verification of RISC-V cores". MA thesis. Politecnico di Torino, 2018.

[18] Peter Sarson. "Analog Test Engineering". In: *Politecnico di Torino*. 2018.

[19] M. Sonza Reorda and P. Bernardi. "Effective Screening of Automotive SoCs by Combining Burn-In and System Level Test". In: *IEEE* (2019).

[20] M. S. Sonza Reorda, F. Corno, and G. Squillero. "RT-level ITC'99 benchmarks and first ATPG results". In: *IEEE Design & Test of Computers* 17.3 (2000), pp. 44–53. DOI: 10.1109/54.867894.

[21] A. Tonda G. Squillero. *Microgp*. Oct. 8, 2019. URL: http://ugp3.sourceforge.net/..

[22] K. G. Verma, B. K. Kaushik, and R. Singh. "Effects of process variation in VLSI interconnects – a technical review". In: *Microelectronics International* 26.3 (2009), pp. 49–55. DOI: 10.1108/13565360910981562.

[23] R. Xie et al. “A 7nm FinFET technology featuring EUV patterning and dual strained high mobility channels”. In: *2016 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2016. DOI: 10.1109/iedm.2016.7838334.

[24] Y. Zorian, S. Dey, and M. J. Rodgers. “Test of future system-on-chips”. In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD - 2000. IEEE/ACM Digest of Technical Papers (Cat. No.00CH37140)*. IEEE, 2000. DOI: 10.1109/iccad.2000.896504.