High-Level Test of Electronic Systems

Collaudo ad alto livello di Sistemi Elettronici

Gianluca Cumani
Dottorato di Ricerca
Ingegneria Informatica e dei Sistemi – XV Ciclo
Politecnico di Torino

Index

Index ..... **I**
Figure Index ..... **III**
Table Index ..... **IV**
1 Introduction **1**
1.1 VLSI Testing 1
1.2 Test generation 2
1.3 RT-Level Test 4
1.4 Microprocessor Test 8
2 RT-Level Testability **12**
2.1 RT-Level Fault Model 12
2.1.1 RT-Level Single-bit Stuck-at Fault Model 13
2.1.2 Gate-Level Correlation Rules 15
2.1.2.1 Rule A 15
2.1.2.2 Rule B 16
2.1.2.3 Rule C 16
2.1.2.4 Examples 17
2.2 RT-Level Fault Simulation Techniques 19
2.2.1 Fault Simulation Environment 20
2.2.1.1 General architecture 20
2.2.1.2 The Fault List 20
2.2.1.3 The Fault Simulator 21
2.2.1.4 Fault Injection Strategy 22
2.3 RT-Level Fault Model Feasibility 25
3 High-Level ATPG **29**
3.1 ARPIA 31
3.2 Fault Model 32
3.3 Fault Simulation Technique 33
3.4 Algorithm 33
3.5 Experimental Evaluation 36
4 Microprocessor Test **40**

4.1 A Semi-Automatic Test Program Generation Methodology 41
4.1.1 Test Strategy 42
4.1.2 Test Program Generation 44
4.1.3 Algorithm 45
4.1.4 Genetic Algorithm 48
4.1.5 Experimental Evaluations 49
4.1.6 Methodology Limits 52
4.2 An Automatic Test Program Generation Methodology 54
4.2.1 Test-program Generation 56
4.2.2 Program Evaluation 56
4.2.3 Assembly-Level Test-Program Induction 57
4.2.4 Directed-Acyclic-Graph Representation 58
4.2.5 Program Induction 62
4.2.6 Auto Adaptation 64
4.2.7 Experimental Evaluation 65
4.2.7.1 i8051 66
4.2.7.2 DLX/pII 68
4.2.7.3 LEON P1754 72
5 Conclusions 75
6 References 79

Figure Index

Figure 1: System-On-a-Chip....8
Figure 2: RT-level Single-bit Stuck-at Fault Example....14
Figure 3: Fault Simulator Algorithm....21
Figure 4: Correlation without rules....27
Figure 5: Correlation with rules....27
Figure 6: Phase One Pseudo Code....34
Figure 7: Phase Two Pseudo Code....35
Figure 8: Pseudo-code of the macro for the ADD instruction.....42
Figure 9: RT-level Single bit Stuck-at Fault Example....43
Figure 10: Test Program Generation Architecture....45
Figure 11: Control-dependent fault detection phase pseudo-code.....46
Figure 12: Data-dependent fault detection phase pseudo-code. 47
Figure 13: Experimental setup for comparison purposes. 50
Figure 14: Pipeline Effects....55
Figure 15: System Architecture 56
Figure 16: A sequential instruction....59
Figure 17: DAG and Instruction library....60
Figure 18: Conditional Branch....61
Figure 19: Mutation Operands. 63
Figure 20: Crossover Operand. 64
Figure 21: Auto Adaptation. 65

Table Index

Table 1: Rule A application 17
Table 2: Rule B application 18
Table 3: Rule C application 18
Table 4: Influence of the rules on fault coverage 25
Table 5: Influence of the rules on correlation 26
Table 6: ARPIA Result 36
Table 7: Comparison with ARTIST and ARPIA without Evolutionary Algorithm 37
Table 8: Phase Two Effectiveness 38
Table 9: 8051 description characteristics. 49
Table 10: Genetic Algorithm Parameters. 50
Table 11: Test Program generation from RT-level description. 51
Table 12: Test Program generation from gate-level description. 51
Table 13: i8051 Test Program Generator Parameters 67
Table 14: i8051 Experimental Results 67
Table 15: DLX Summary 69
Table 16: DLX Random Approach Summary 70
Table 17: DLX Statement Coverage Breakdown 71
Table 18: DLX Toggle Activity Breakdown 72
Table 19: LEON Statement Coverage summary 73
Table 20: LEON Statement Coverage Breakdown 73

1 Introduction

1.1 VLSI Testing

In the last decades, the advances in Very Large Scale of Integration (VLSI) semiconductor technologies have caused the incredible development of electronic systems. The reduction of device sizes makes now possible to fit increasingly larger number of transistors onto a single chip. However, as chip density increases, the probability of defects occurring in a chip increases as well.

Design and manufacturing failures are becoming an important part of the microelectronics business, where complexity is growing rapidly. Failures can occur at several points of a product life cycle, such as technology or product development and qualification, yield learning, reliability improvement, system manufacture, and field application.

The quality, reliability and cost of the product are directly related to the intensity/level of testing of the product. For this reason, Integrated Circuits (ICs) testing has gradually shifted from the final fabricated ICs to the design stage and many Design for Testability (DFT) techniques have been developed to ease the testing process [PJAV02].

The functionality of a combinational circuit can be verified by exhaustively applying all possible input patterns to its inputs. The behavior of a sequential circuit, instead, is based on the applied sequence of the input patterns, so, their functionality can’t be verified by applying all possible input patterns. Testing involves generating test patterns, applying them to the circuit, then analyzing the output response. Testing falls into a number of categories depending upon the intended goal.

Structural test looks for faults that can occur in the physical structure of a manufactured component (e.g., stuck-at faults).

Go/no go test determines whether or not a manufactured component is functional. This test gets executed on every manufactured die and has a direct impact on the cost (test equipment cost, testing time, etc.), so it should be as simple and swift as possible.

Parametric test checks a number of parameters, such as: voltage levels, current levels, noise margin, propagation delays, capacitive coupling or cross talk and maximum clock frequencies. These parameters are tested under a variety of working conditions, such as temperature and supply voltage.

Path-delay or *transition-delay* techniques are extensions of stuck-at faults testing. They have been developed to catch timing related defects.

I_{DDQ} *testing* checks the most common defects in Complementary Metal Oxide Semiconductor (CMOS) technology caused by the shrinking geometry and thinner gate dielectric, such as bridging between different tracks and gateoxide shorts.

Diagnostic test, finally, is used during the debugging of a chip or board and tries to identify and locate the fault in a failing chip or board.

1.2 Test generation

Physical circuit defects are often modeled as logic faults [PJAV02]. The assumption being that any manufacturing defect will translate itself into an erroneous logic value at a specified time. This assumption is called *fault modeling*. This makes the problem of fault analysis independent of the technology. Fault modeling analyzes the circuit's behavior in the presence of faults caused by physical defects or environmental influences and provides a basis for the fault simulation and test generation.

Test pattern validation is the process of determining how well a test pattern meets the fault coverage requirements. The fault coverage is defined as the ratio of the number of faults that are detected and the total number of faults in the assumed fault universe. Fault coverage computation is a very important step in the testing process. It provides a measure of adequacy for a given test set.

The functionality of a combinational circuit can be verified by exhaustively applying all possible input patterns to its inputs. For an N-input circuit, this requires the application of 2^N patterns. In a sequential circuit, the output of the circuit depends not only on the inputs applied, but also on the state of the internal memory elements of the circuit. An exhaustive test of a sequential circuit requires the application of 2^{N+M} test patterns, where N and M are the number of inputs and registers/flip-flops of the circuit respectively. As N and M increase, the test length expands exponentially. To reduce the test pattern length, an alternative approach is required.

Test generation is the process of determining a set of stimuli necessary to test a circuit. The computational cost of the test generation depends on the complexity of the method. A test for a fault can be found by trying various input patterns until one gives a different output so that the considered fault is found in the circuit.

Random Test Generation (RTG) is a simple process that involves only the generation of random vectors. However, to achieve a high-quality test we need a large set of random vectors. RTG works without taking into account the function or the structure of the circuit to be tested.

In contrast, *Deterministic Test Generation* produces tests by processing a model of the circuit. Deterministic test generation can be fault-independent or fault oriented. Fault-independent test generation works without targeting individual faults. One early approach in test generation was using the concept of boolean difference for generating the test patterns. The boolean difference approach can be characterized as algebraic. It manipulates circuit equations to generate test patterns. In the context of VLSI systems, test generation by algebraic methods is too time-consuming; hence, the algorithmic approaches are being used.

In fault-oriented process, tests are generated for specified faults of a fault universe. The test generation methods use the topological gate-level description of the circuit. The algorithmic approaches also use various mechanisms to trace sensitive paths to propagate fault effects to primary outputs. They then back trace to the primary inputs and assign logic values based on conditions set at the forward fault propagation stage. The test generation methods generate test vectors for one fault at a time. However, as the number of internal nodes in Circuit Under Test (CUT) increases,

the computation time in generating the input test patterns becomes enormous. In VLSI systems the time required to generate the test patterns for the whole system to get full fault coverage is quite high, for this reason the problem is partitioned into smaller parts. The solution for each part is obtained and then combined into a solution for the whole problem.

Traditionally, Automatic Test Pattern Generators (ATPGs) target the test generation problem at the logic level. However, they can require large amounts of computing time and resources to generate tests for even moderately sized sequential circuits.

Computation time can be drastically reduced by mapping the Register Transfer Level (RTL) signals of the circuit being tested to its corresponding gate-level nets. Several methods have been reported which automatically use functional RTL descriptions of the circuit that target detection of stuck-at faults in the circuit at the logic level.

In recent years the ASIC design flow experienced radical changes. The ASIC design flow is rapidly moving towards higher description levels. This increasing demand for tools enabling the design of digital circuits at high levels of abstraction already pushed the development of synthesis and simulation technologies.

At the same time technology advances allow integrating on a single chip entire system, including memories and peripherals. The test of these devices is becoming a major issue for manufacturing industries. For this reason we need an efficient and versatile methodology for inducing test-programs starting from higher level descriptions.

1.3 RT-Level Test

The common practice is to design, simulate and synthesize huge ASICs entirely at the RT-level. For this reason high-level design for testability, testable synthesis and test pattern generation are increasing their industrial relevance. During ASIC development, designers would like to be able to foresee testability before starting the logic synthesis process. However, despite many efforts in high-level design for testability, testable synthesis and test pattern generation, tackling testability

at high levels is still an unsolved problem. In addition, there is an ever-increasing demand on reducing time to market. With complexity skyrocketing and such a competitive pressure, designing at high levels of abstraction has become more of a necessity than an option.

The increasing complexity of electronic components may be faced only by boosting designer productivity through a gradual shift towards higher abstraction levels and to significant amounts of design reuse. Nowadays, most digital ASICs are designed at the RT-level, thanks to the availability and maturity of HDL synthesis tools. Other design activities, such as power estimation and testing, are lacking behind this trend, and are still performed mainly at the gate-level.

In recent years, several research activities contributed to pushing testability related issues to the RT-level, including the proposal of several fault models [DGKe96] [TAZa99] [RiUc96], the development of fault simulators [FiFu00] [FDKe98] or testability analyzers, and of some test pattern generators [FFSc98] [CSSq00a] [FADe99].

The hardest *theoretical barrier* to the diffusion of test-related tools at the RT-level is the lack of widely accepted fault models. Several variants of high level faults (or testability metrics) have been proposed, and their relationship with stuck-at faults has been shown, either experimentally or theoretically, but such results are generally limited to some specific class of circuits. No single fault model is universally accepted, since no comprehensive and general results, valid for all classes of circuits, are known yet.

Most fault modeling approaches rely on high-level fault models for behavioral HDL descriptions which have been developed by the current practice of software testing [Beiz90] and extending them to cope with hardware descriptions. In this sense, a high-level fault model corresponds to a *metric* that measures the goodness of a given sequence of input vectors.

One of the most used fault models is the *observability enhanced statement coverage* metric proposed in [DGKe96] and [FDKe98]. This fault model requires that all statements in the VHDL description are executed at least once, and that their effects are propagated to at least one primary output. Propagation is modeled

implicitly, by determining whether the faulty statement may influence the output values but without hypothesizing any specific faulty value: in some cases, heuristics are needed to resolve non-determinism, and the meaningfulness of the resulting fault coverage is affected by these approximations. While this approach can be fruitfully exploited for test pattern generation [FADe99] [CSSq00a], for fault simulation we need more accurate results.

One of the most important *technical barriers* is the lack of efficient fault simulators, once a fault model is chosen. Fault simulation algorithms for RT-level descriptions are known since more than a decade, even if they mainly target structural descriptions rather than behavioral ones, but commercial tools usually do not include these capabilities. Classical algorithms are difficult to integrate in HDL simulators, mainly due to the complexity and to the several peculiarities of HDL languages. Until some fault model becomes widely accepted, this situation is not likely to change, because CAD vendors have no good reason to invest yet.

In this work an approach is illustrated, that allows fault simulation at the RT-level of VHDL descriptions, by interacting with a standard commercial VHDL simulator. The approach is based on exploiting debugging mechanisms inherent with the chosen VHDL simulator and exposed through the scripting language interface, such as breakpoints, script and TCL programming, and signal traces, and allows an accurate and fast simulation of faulty behaviors through a minimally invasive procedure. Other approaches were formerly proposed in [RiUc96], where for each fault a newly modified VHDL description was built, compiled, and simulated, and in [FiFu00], where a single modified VHDL model foresaw all the possible single and multiple fault locations and values. In our approach, VHDL descriptions are never modified, so that simulation always proceeds at full speed for all the circuit except the fault insertion point, and more complex VHDL constructs can be accepted at little implementation cost.

In this work we thus adopt a particular instantiation of the observability enhanced statement coverage metric, and in particular we model *single stuck-at bit faults* on all assignment targets of the executed statements that respect a defined set of rules. With this choice, a concrete faulty behavior is simulated, and fault propagation

can therefore be performed exactly, by computing the faulty machine evolution. This fault model implies observability enhanced statement coverage, since it models one of the possible fault classes on executed statements. We also define a series of rules to identify redundant faults in the fault list, obtained using the proposed fault model, in order to increase the correlation between the RT-level fault coverage and the Gate-level one. Redundancies identification is based on the reduction of the RT-level fault list taking into account analyzing the optimizations of the synthesis process, in order to eliminate faults corresponding to part of the logic optimized away by the synthesizer.

Another *technical barrier* is the lack of efficient algorithms to generate effective test signals. As a matter of fact, at the present time even *simulating* RT-level test signals is a challenging task. Fault simulation algorithms for RT-level designs are known since more than a decade [ABFr90], but commercial tools usually don't include these capabilities. Furthermore, classical algorithms are difficult to integrate in simulators, mainly due to the complexity and to the several peculiarities of hardware description languages. Some prototypical fault simulators were proposed [FiFu00] [FDKe98], but until some fault model becomes widely accepted EDA industries have no good reason to invest, and this situation is not likely to change.

Even so, researchers and pioneering design groups already need test signals on their RT-level designs. Many generators were proposed. Nevertheless, since any attempt of backward justification must take into account all structural, behavioral and timing specifications [FFGS99], traditional algorithms are almost unusable. Researchers sometimes achieved good results, but they were generally limited to specific classes of circuits. However, interesting successful results have been reported using evolutionary algorithms [CSSq00a]. These approaches exploit natural evolution principles to drive the search of effective patterns within the gigantic space of all possible signal sequences. Evolutionary heuristics begin to appear a reasonable alternative to traditional techniques.

1.4 Microprocessor Test

Technology advances allow integrating on a single chip an entire system, including memories and peripherals. This new kind of devices is called *Systems-On-a-Chip* (SOCs) and often includes complex cores, like microprocessors, MPEG encoders/decoders and other dedicated blocks (Figure 1). Such cores are usually provided by third parties and designers may include them into SOCs rather easily and quickly.

A core is a highly complex logic block. It is predictable, reusable, and fully defined in terms of its behavior. System designers can purchase cores from core-vendors and integrate them with their own user-defined logic to implement SOCs more efficiently. Core-based SOCs have significant advantages: core exploitation reduces the number of required discrete components, minimizing the total size and cost of the end-product; furthermore, it greatly reduces time-to-market because of the involved design re-use.

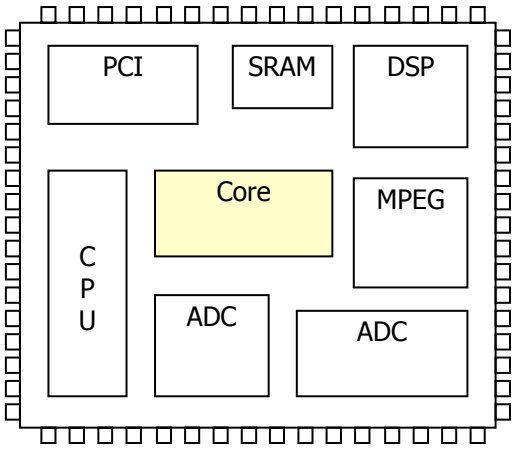

Figure 1: System-On-a-Chip

Microprocessor cores represent an important and widespread class of macros, and they are a major challenge in the test arena. Not only is their complexity always increasing, but also their specific characteristics intensify all existing difficulties. A microprocessor embedded inside a SOC is harder to test since it might be harder to

control and its behavior may be harder to observe. And design-for-testability structures might be harder to insert, too.

Regarding microprocessors, test has traditionally been performed resorting to functional approaches based on exciting functions and resources. The most canonical one is described in [ThAb80]; however, this methodology involves a high amount of manual work performed by skilled programmers, and does not provide any quantitative measure about the attained gate-level fault coverage.

[ShAb98] proposes a methodology to synthesize a self-test program for stuck-at faults. The approach generates a sequence of instructions that enumerates all the combination of the operations and systematically selects operands. However, users need to determine the heuristics to assign values to instruction operands to achieve high stuck-at fault coverage. In some cases, this might not be a trivial task.

[ChDe00] proposes DEFUSE, a deterministic method to generate test programs able to reach good fault coverage on the ALU of a microprocessor, and to compact the result. The approach is very effective with combinationally testable parts (i.e., ALUs), but shows some limitation when hard-to-test sequential modules (e.g., control units) are addressed. On the other hand, [BaPa99] is based on generating random sequences of instructions. It is able to attain a fairly high level of fault coverage, however it assumes that all instructions are single-cycle and buses are never floating. Both approaches require the insertion of BIST circuitry.

On a microprocessor core, however, test solutions requiring massive chip-level changes might not be exploitable. In particular, any solution based on a scan approach may be inapplicable. First of all, the insertion of SOC-level test architectures for allowing external access to the scan chains might be unpractical or incompatible with other DFT structures. Furthermore, system designers may dislike scan methodologies since they do not allow at-speed tests and they could degrade performance.

Any test solution requiring the application of binary values directly to the microprocessor pins may potentially cause problems to SOC designers. The ideal strategy should consist in a mere assembly program and not rely on any special test point to force values or observe behaviors. Such test program could be loaded in

RAM (e.g., resorting to DMA or other mechanisms), and executed to test the core. A minimum effort could be needed to extract test result.

The requirements for such a test solution are analogous to the ones described in [PMNo99] and [CSSV01]: a RAM memory of sufficient size should be available on the SOC and easily accessible from the external. In this way, an ATE can load into the memory the test program when required, and the processor core can execute it. Test execution is always performed at-speed, independently on the speed of the mechanisms used for loading the RAM and checking results.

Regarding test program generation, [BiMa95] proposes interesting techniques for efficient compilation of self-test programs. But they left the responsibility for generating the self-test programs to the test engineers.

Tackling verification, [AABH99] proposed a technique where the processor itself generates test at run-time by self-modifying code. Similarly, [UBSh99] showed a method for generating instruction sequences for validating the branch prediction mechanism of the PowerPC604. Generated sequences are very effective, but methodologies exploit a deep knowledge of the target processors and cannot be easily applied on general designs.

[CSSV01] proposes a semi-automated approach to test program generation based on a library of macros those parameters are chosen by a genetic algorithm. The approach is shown able to attain reasonable fault coverage (85%) on a common microprocessor core and requires no additional hardware or scan structures. However, test generation relies on a library carefully compiled by experts.

[KPGZ02] describes a methodology that allows devising an effective test program for a microprocessor core. However, the method requires that test engineers create deterministic test patterns to excite the entire set of operations performed by each component of the core.

Other possible approaches include the cross-compilation of available high-level routines. However, despite the effortlessness, this is not a good solution. Due to the intrinsic nature of the algorithms and of compiler strategies, these programs are seldom able to excite all functionalities and do not take into account observability.

Although more effective and easier to generate, also random programs neglect observability and will hardly detect hard-to-test faults. Moreover, their exploitation could require huge memory space and overlong test times.

This document is organized as follows. Chapter 2 focuses on fault model and fault simulation at the RT-level, and aims at exploiting the capabilities of VHDL simulators to compute faulty responses. The simulator was implemented as a prototypical tool, and experimental results show that simulation of a faulty circuit is no more costly than simulation of the original circuit. The reliability of the fault coverage figures computed at the RT-level is increased thanks to an analysis of inherent VHDL redundancies, and by foreseeing classical synthesis optimizations. A set of "rules" is used to compute a fault list that exhibits good correlation with stuck-at faults.

Chapter 3 describes a new simulation-based evolutionary test generator (ARPIA) that adopts the innovative high-level fault model that enables efficient fault simulation and guarantees good correlation with gate-level results described in Chapter 2. The approach exploits an evolutionary algorithm to drive the search of effective patterns within the gigantic space of all possible signal sequences. ARPIA operates on register-transfer level VHDL descriptions and generates effective test patterns. Experimental results show that the achieved results are comparable or better than those obtained by high-level similar approaches or even by gate-level ones.

Chapter 4 describes two efficient and versatile approach to test-program generation based on an evolutionary algorithm. The described methodologies are able to tackle from microcontroller to complex pipelined designs.

2 RT-Level Testability

2.1 RT-Level Fault Model

One of the hardest *theoretical barriers* to the diffusion of test-related tools at the RT-level is the lack of widely accepted fault models. Several variants of high level faults (or testability metrics, as they are sometimes called) have been proposed, and their relationships with stuck-at faults has been shown, either experimentally or theoretically, but such results are generally limited to some specific class of circuits (some approaches target control-dominated circuits [CSSq00a], others are more suited to data-dominated ones [FADe99], or to circuits with few interactions with the environment [FiFu00], and so on). No single fault model is universally accepted, since no comprehensive and general results, valid for all classes of circuits, are known yet.

One of the most used fault models is the *observability enhanced statement coverage* metric proposed in [DGKe96] and [FDKe98]. This fault model requires that all statements in the VHDL description are executed at least once, and that their effects are propagated to at least one primary output. Propagation is modeled implicitly, by determining whether the faulty statement may influence the output values but without hypothesizing any specific faulty value: in some cases, heuristics are needed to resolve non-determinism, and the meaningfulness of the resulting fault coverage is affected by these approximations. While this approach can be fruitfully exploited for test pattern generation [FADe99] [CSSq00a], for fault simulation we need more accurate results.

In this work we thus adopt a particular instantiation of the observability enhanced statement coverage metric, and in particular we model *single-bit stuck-at faults* on all assignment targets of the executed statements that respect a defined set of

rules. With this choice, a concrete faulty behavior is simulated, and fault propagation can therefore be performed exactly, by computing the faulty machine evolution. This fault model implies observability enhanced statement coverage, since it models one of the possible fault classes on executed statements. We also define a series of rules to identify redundant faults in the fault list, obtained using the proposed fault model, in order to increase the correlation between the RT-level fault coverage and the Gate-level one. Redundancies identification is based on the reduction of the RT-level fault list taking into account analyzing the optimizations of the synthesis process, in order to eliminate faults corresponding to part of the logic optimized away by the synthesizer.

2.1.1 RT-Level Single-bit Stuck-at Fault Model

Fault models taken from software-testing [Beiz90] have three main advantages: they are well known and quite standardized; they require little calculations, apart from the complete simulation of the fault-free system; and they are already embedded in some commercial tools. However, while such metrics may be useful to validate the correctness of a design [CSSq00], they are usually inadequate to foresee the gate-level fault coverage with high degree of accuracy.

To improve accuracy, some researchers extended software metrics to cope with the peculiarities of hardware descriptions. Fallah *et al*. [FADe99] [FDKe98] proposed *Observability-Enhanced Statement Coverage*. They define the concept of *tag* as the possibility that an incorrect value is computed at a given location. Different tags are first *injected* in any possible location and then *propagated* during the simulation. The observability-enhanced statement coverage metric computes the number of tags that reach an observable circuit output when the test pattern is applied.

We adopt observability-enhanced statement coverage and we refine it by using explicit *RT-Level single-bit stuck-at*'s instead of tags. An RT-level single-bit stuck-at fault is defined as a single-bit stuck-at in the effect of an RT-level assignment operation: when a fault is present, the affected object (signal or variable target of an assignment statement) loads the correct value, except for one bit that remains stuck to 0 or 1.

As in [DGKe96], faults are *single* and *permanent*: only one fault is inserted at a time and the fault effect is present during the whole simulation. The RT-Level single-bit stuck-at fault model does not explicitly consider control-flow faults, such as *stuck-at-true* or *stuck-at-false*, as [RiUc96] does.

In more details:

- For bit signals or variables, the fault forces the value of 0 or 1 regardless the actual value. No other values (e.g., 'Z') are considered.
- In bit vector signals or variables, each single element is considered separately as a bit.
- Integer signals or variables are translated into the equivalent bit vectors according to synthesis conventions. Range checks are neglected in the resulting vector.
- Enumerated signals or variables are translated into integers and bounds are ignored. If a fault forces an enumerated object to an illegal value causing the simulator to stop, it is marked as detected.
- Faults on input ports are taken into account by considering the operation of setting an external value to a primary input as an implicit assignment operation.
- Concurrent expressions are translated into their equivalent processes.
- VHDL hierarchy is flattened, thus a process instantiated more than once is seen as multiple processes.

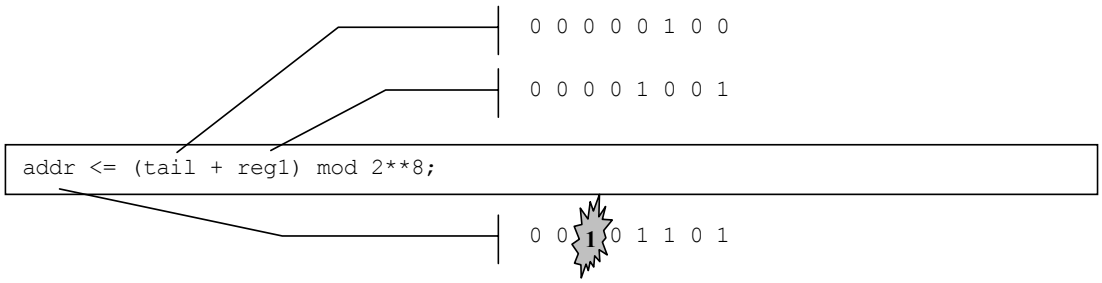

Figure 2: RT-level Single-bit Stuck-at Fault Example

Figure 2 shows the example of an RT-level single-bit stuck-at fault. The fault affects the third bit of the assignment operation, and modifies the result of the expression, after it has been computed and before it is assigned to the target signal. The faulty signal is updated as usual, according to VHDL propagation rules, but with a faulty value. Other assignments of the same signal are assumed to be fault-free, since stuck-at faults on the same signal but on different statements are considered different.

2.1.2 Gate-Level Correlation Rules

During synthesis the RT level VHDL description is optimized in order to create an efficient gate-level design. The optimization process analyzes the VHDL description and simplifies all logic eliminating redundancies. In this phase some RT level stuck-at faults lose their correspondent gate-level faults. The elimination of these gate-level faults generates a discrepancy between RT and gate-fault coverage figures. In order to prevent this discrepancy is necessary to identify which parts of the logic described at the RT-level disappear during the optimization phase of the synthesis process and to eliminate the associated faults from the fault list.

The logical elements that are eliminated during optimization process are:

- assignment of constant values to a signal or variable;
- signals or variables with only few bits actually used in the system.

Another discrepancy is introduced from the different approach of the RT-level fault simulation and the gate-level one. In the gate-level fault simulation the reset signal of the system is considered *fault-free*, so no faults are simulated in this part. To prevent the difference between the two fault simulation methodologies, the RT-level faults concerning the VHDL part executed only when the reset signal is active must be eliminated.

Those concepts are formalized by the following set of rules.

2.1.2.1 Rule A

When a constant value is written in a variable or a signal, all the logic that was used to accomplish the constant part of the operation is reduced to a set of wires

connected directly to the flip-flop. For this reason the only faults identifiable at gate-level are the ones stuck-at with a value different from the constant one.

In order to identify such useless faults it's necessary to:

- identify faults concerning variables or signals destination of an assignment instruction;
- determine if all or some of the bits of the second term of the instruction are constant, or an operation with a constant result, and calculate its value;
- eliminate the faults on the bit whose stuck-at value is equal to the constant value at the same bit.

2.1.2.2 Rule B

Sometimes in the VHDL description there are variables or constants that have only a few bits really useful for the system. During the optimization process the size of this variables or signals are reduced to the number of bit really useful. All the faults concerning the erased bits must be eliminated from the fault list.

In order to identify such useless faults is necessary to:

- determine variables or signals that are only used in conditional expressions;
- determine for any conditional expression if the second term is a constant, or an operation between constants, and calculate its value;
- calculate the subset of bit values common to the various constants;
- eliminate the faults on the bits whose stuck-at value is equal to the subset value at the same bit;

2.1.2.3 Rule C

At gate-level all the logic connected to the reset signal is considered *fault-free* for this reason all the faults concerning those instructions must be eliminate from the fault list. This hypothesis is needed due to the limitations of VHDL models, that cannot describe the correct behavior of a circuit if it fails to be correctly initialized (unknown valued don't propagate correctly across conditional statements). To provide

a meaningful comparison, we do not consider faults in the reset logic, either at the RT- or gate- levels.

In order to identify such useless faults is necessary to:

- determine the part of the VHDL source executable only when reset signal is active;
- eliminate faults concerning variables or signals destination of an assignment in this part;

2.1.2.4 Examples

To illustrate some examples of application of the above rules, see Tables 1 to 3. The tables report a sample VHDL statement or fragment, the information about faults to be injected (source line, signal or variable name, bit position of the fault and stuck-at value), as well as the indication whether the application of the rule *eliminated* the fault from the fault list.

VHDL instruction	eliminated	source line	name	bit	stuck-at value
State<=000;	**Yes**	**29**	**STATE**	**0**	**0**
	No	29	STATE	0	1
	Yes	**29**	**STATE**	**1**	**0**
	No	29	STATE	1	1
	Yes	**29**	**STATE**	**2**	**0**
	No	29	STATE	2	1

Table 1: Rule A application

In Table 1, the assignment of the constant “000” to signal “State” is considered. All stuck-at-0 faults on the bits of State, in injected at this instruction, are untestable since they are not excited. According to Rule A, they are excluded from the fault list, while stuck-at-1 faults will be injected.

VHDL source	eliminated	source line	name	bit	stuck-at value

temp= bit_vector(2 downto 0);
……..
temp := DATA_IN+REG
……
if (**temp >= 0**) then
	Yes	**29**	**TEMP**	**0**	**0**

	Yes	**29**	**TEMP**	**0**	**1**
	Yes	**29**	**TEMP**	**1**	**0**
	Yes	**29**	**TEMP**	**1**	**1**
	No	29	TEMP	2	0
	No	29	TEMP	2	1

Table 2: Rule B application

In Table 2 a different case is considered, where the value of variable "temp" is only used in a greater-than-zero comparison. In such a situation, only the bit sign of the variable is significant, and all faults on lower order bits are deleted by Rule B since they are not observable. As a matter of fact, synthesis tools are able to detect this situation, and do not generate the logic associated to lower order bits: we are effectively predicting that some RT-level faults have no physical meaning since no gate-level equivalent will be synthesized.

VHDL source	eliminated	Source line	name	bit	stuck-at value
reset='1' then **state:=a;** elsif state:=b;	**yes**	**32**	**STATE**	**0**	**0**
	yes	**32**	**STATE**	**0**	**1**
	yes	**32**	**STATE**	**1**	**0**
	yes	**32**	**STATE**	**1**	**1**
	no	34	STATE	0	0
	no	34	STATE	0	1

Table 3: Rule C application

Finally, Table 3 shows an application of Rule C, where all RT-level faults dominated by the reset signal are deleted.

2.2 RT-Level Fault Simulation Techniques

Fault simulation at the RT-level is an open issue that is expected to gain high industrial relevance with the advent of high level testability flows and that is proven to yield good coverage with actual defects [SGTT00].

One of the most important *technical barriers* is the lack of efficient fault simulators, once a fault model is chosen. Fault simulation algorithms for RT-level designs are known since more than a decade, even if they mainly target structural-style descriptions rather than behavioral-style ones, but commercial tools usually don't include these capabilities. Classical algorithms are difficult to integrate in HDL simulators, mainly due to the complexity and to the several peculiarities of HDL languages. Until some fault model becomes widely accepted, this situation is not likely to change, because CAD vendor have no good reason to invest yet.

In this work an approach is described, that allows fault simulation at the RT-level of VHDL descriptions, by interacting with a standard commercial VHDL simulator. The approach is based on exploiting debugging mechanisms inherent with the chosen VHDL simulator and exposed through the scripting language interface, such as breakpoints, script and TCL programming, and signal traces, and allows an accurate and fast simulation of faulty behaviors through a minimally invasive procedure. Other approaches were formerly proposed in [RiUc96], where for each fault a newly modified VHDL description was built, compiled, and simulated, and in [FiFu00], where a single modified VHDL model foresaw all the possible single and multiple fault locations and values. In our approach, VHDL descriptions are never modified, so that simulation always proceeds at full speed for all the circuit except the fault insertion point, and more complex VHDL constructs can be accepted at little implementation cost.

2.2.1 Fault Simulation Environment

2.2.1.1 General architecture

In order to verify the feasibility of the proposed fault simulation technique, we developed a prototype implementation of a Fault Simulator that, starting from a VHDL description at the RT-level, a Fault List of single-bit stuck-at faults and a Test Pattern, creates a list of detected and undetected faults.

To perform Fault Simulation we use a serial fault simulation strategy, and we simulate the good and each faulty machine, comparing their outputs. To run the simulations, the Test Pattern is first transformed to a set of commands that force the correct waveform for input signals, and the Fault List is transformed to a set of script commands for injecting faults during simulation.

Starting from the above considerations we developed Fault Detector System composed of the following elements:

- **Fault List Generator**: this tool extracts information (signal/variable names, hierarchy, type and source code line) from the analysis of VHDL Source code and creates the Fault List based on the proposed fault model.
- **Fault Simulator**: this tool is composed of a set of routines interacting with the VHDL simulator. It simulates the circuit described by the VHDL Source using the Test Pattern and injects the faults present in the Fault List, creating a list of Detected Faults.

2.2.1.2 The Fault List

As a preliminary step, for each design we extract a complete list of faults, by analyzing the VHDL source code and enumerating faults on input signals and on internal signals and variables. We analyze the code with the help of the LEDA VHDL*Verilog System database and of ModelSim EE 5.1g scripts, and we obtain input signal names and types and assignment instructions with their VHDL source lines. By parsing assignment instructions we determine the signal or variable name and type. For hierarchical descriptions, the above analysis is preceded by flattening of the hierarchy, where multiply instantiated processes are considered different.

Information obtained by the VHDL source code analysis is collected in the Fault List. For each bit of each signal and variable we generate two Fault List entries, for the stuck-at '1' and stuck-at '0' faults, containing the above information. Each fault is described by a tuple composed of: VHDL source file name, source line (not relevant for input faults), the target type (input, signal or variable), target hierarchical name, bit position, stuck-at value and some fault detection information. After simulation, each entry is updated with the indication of fault status (detected or undetected) and the number of the pattern detected it.

2.2.1.3 The Fault Simulator

The Fault Simulator is the core of the Fault Detector Architecture. This part of the tool injects faults according to a serial fault simulation methodology: for each fault, the entire test pattern is simulated, and outputs are compared. Several optimizations can be implemented over this basic scheme, and will be the subject of further work, while the current implementation already proves the feasibility of the approach. A pseudo code description of the Fault Simulator is reported in Figure 3.

```
ReadFaultList();
ReadTestPatterns();
InitializeSimulator();
/* simulate the good machine */
Simulate(good);
StoreOutputs(good);
for(each fault)
{
  /* simulate the faulty machine */
  InjectFault(fault);
  Simulate(fault);
  if (CompareOutputs(good, fault) == DIFFER)
    UpdateFaultList(fault, DETECTED);
  else
    UpdateFaultList(fault, UNDETECTED);
}
```

Figure 3: Fault Simulator Algorithm

2.2.1.4 Fault Injection Strategy

The core of the Fault Simulator is the Fault Injection procedure. Several different approaches for injection of permanent faults in VHDL descriptions are possible, some of which have already been proposed in the literature:

- **Changing the VHDL code:** original VHDL instructions are enriched by the code necessary to inject the fault and new input signals are added to control fault injection [FiFu00]. This technique significantly slows down simulation, because the additional source lines are always simulated, also when they are not used to inject the fault.
- **Modifying the simulator:** the code necessary to inject and detect faults is added into simulator source code. This technique is probably the fastest fault injection methodology, and promises to simulate each faulty machine as fast as the fault free circuit, and is extremely powerful, because one may change any parameter or register during simulation. The problem of this technique is the availability of the source code of a good simulator.
- **Interacting with the simulator**: faults are injected through the simulator user interface using simulation commands. This technique is less powerful than modifying the simulator, but during simulation it is nearly as fast. In fact, no additional source code is present and commands are active only when the fault is injected.

Our fault injection system belongs to the third methodology. Fault injection is made possible by creating routines that *change* the target signal/variable bit value during simulation, using the Simulator Scripting Language (TCL), when a given target assignment instruction is executed.

The chosen fault injection methodology must face various issues derived from the fault model, from VHDL semantics and from the simulator itself.

The chosen *fault model* considers both input signals and internal signals or variables: while the fault model definition treats them uniformly, from the implementation point of view they are different. The former ones, subject of no

assignment instruction, do not correspond to a source code line identifiable as target during the simulation, while the latter may be written several times in the VHDL description, thus preventing a statically "forced" assignment of the faulty bit. Two different fault injection methods must therefore be used: one for input signals and one for internal signals or variables. Input signals can be modified before simulation starts, but internal variables and signals must be changed during simulation, whenever the target assignment instruction is executed.

VHDL semantics specify that signals change at wait statements and variables change immediately. Consequently internal signals and variables must be treated in different ways during fault injection. Variables can be changed immediately after the execution of an assignment instruction; signals instead may be changed only just before the execution of the wait statement (or of the last line of the process, if the wait statement is implicit).

The *simulator* limits interaction to the commands exposed through the scripting user interface. All the commands necessary to inject the fault must therefore respect the syntax and the timing of the simulator. Specifically, the ModelSim simulator accepts simulation commands and TCL routines. In our case, fault injection is performed through insertion of appropriate breakpoints at target instructions.

Due to the above restrictions, we use two different approaches to inject faults: *pre-simulation fault injection* for input signal faults and *run-time fault injection* for internal faults. Fortunately, no distinction needs to be made concerning the data type of the involved signals and variables, since the simulator interface allows us to treat all objects as bit vectors, regardless of their original type (bit, bit vector, integer, enumerated …).

Pre-simulation fault injection consists of changing the target bit value of an input signal before simulation starts (during the waveform definition phase) forcing it at the stuck-at value. *Pre-simulation fault injection* is fast as a normal simulation because no delay is added to inject faults.

Run-time fault injection consists of changing the target bit value of the assignment instruction selected during the simulation. Breakpoints are used in this case: before simulation starts, a breakpoint is set on the VHDL source line where fault

is located. The Fault List file contains all the information necessary to inject the fault. Embedded in the breakpoint instruction there are two different routines depending on the type (variable or signal) of the assignment instruction target.

If the target is a variable, the instruction is executed, then the given bit of the variable is changed and simulation is continued.

If the target is a signal a more sophisticated *double-breakpoint* technique is needed, to avoid modifying the signal values in advance with respect to VHDL signal propagation semantics. In the double-breakpoint technique, a breakpoint is set at the source code line where the wait statement or the last instruction is placed and simulation continues without modifying anything. This new breakpoint, that will be activated only when the wait statement (explicit or implicit) is about to be executed, forces the given bit of the signal to the stuck-at value and unsets itself before continuing the simulation.

Run-time fault injection using the breakpoint technique slows down simulation by a really negligible amount. In fact, breakpoints are optimized by the simulator and impact on the simulation only when the target statements are executed, and the fault is injected.

2.3 RT-Level Fault Model Feasibility

In order to verify the feasibility of the RT Level Single-bit Stuck-at Fault Model, we used a prototype implementation of a Fault Simulator based on the techniques described in the previous Sections.

circuit	sequence	RT-level Fault Coverage				Stuck-at Fault Coverage
		no rules	rule A	rules A & B	rules A & B & C	
B01	#1	54.23%	95.06%	95.06%	97.37%	98.06%
	#2	50.00%	87.65%	87.65%	90.79%	98.45%
	#3	52.11%	91.36%	91.36%	93.42%	96.51%
B02	#1	46.43%	90.70%	90.70%	94.87%	99.33%
	#2	42.86%	83.72%	83.72%	84.62%	99.33%
	#3	42.86%	83.72%	83.72%	84.62%	99.33%
B03	#1	52.94%	67.61%	67.61%	70.49%	73.84%
	#2	50.37%	64.32%	64.32%	66.67%	74.82%
	#3	48.53%	61.97%	61.97%	63.93%	74.45%
B04	#1	47.16%	55.85%	71.59%	84.22%	91.51%
	#2	49.55%	58.69%	75.23%	88.50%	91.51%
	#3	49.55%	58.69%	75.23%	88.50%	91.92%
B06	#1	32.87%	92.73%	92.73%	98.02%	97.35%
	#2	31.94%	90.91%	90.91%	96.04%	97.35%
	#3	31.94%	90.91%	90.91%	96.04%	97.35%
B07	#1	50.00%	66.93%	66.93%	75.71%	58.28%
	#2	49.25%	65.76%	65.76%	74.29%	57.28%
	#3	52.00%	66.93%	66.93%	79.52%	58.28%
B08	#1	45.75%	60.63%	60.63%	81.90%	82.03%
	#2	51.42%	68.13%	68.13%	91.38%	98.15%
	#3	54.72%	72.50%	72.50%	98.28%	98.26%
B09	#1	38.89%	53.41%	53.41%	59.28%	48.89%
	#2	47.66%	65.46%	65.46%	72.85%	90.56%
	#3	50.00%	68.67%	68.67%	76.02%	91.22%
B10	#1	45.36%	69.90%	69.90%	74.43%	77.70%
	#2	51.32%	79.08%	79.08%	84.66%	92.22%
	#3	51.66%	79.59%	79.59%	85.23%	92.13%
B11	#1	56.33%	64.79%	64.79%	73.91%	79.99%
	#2	54.69%	62.91%	62.91%	71.74%	81.00%
	#3	62.86%	72.30%	72.30%	82.61%	84.52%

Table 4: Influence of the rules on fault coverage

This fault simulation environment allows us to compute fault coverage figures at the RT-level with a minimal CPU time overhead, since the VHDL model of faulty circuits is simulated at the same speed as the fault-free model. The only time penalty is at fault injection time instants, where some breakpoints are activated and TCL commands to modify values are executed.

To perform our experiments we selected a subset of the ITC'99 VHDL benchmarks [CSSq00]. We applied the rules described before to the fault lists extracted for the chosen subset of benchmarks, and fault simulated the optimized one with three samples of input sequences:

- a pseudo-random sequence (#1), consisting of 500 vectors with up to 5 circuit reset commands;
- a test sequence (#2) developed by a simulation-based gate-level developed ATPG [CPRS96];
- a test sequence (#3) generated by a state of the art commercial topological ATPG working at the gate-level.

Table 4 reports detailed results comparing the RT-level fault coverage figures obtained with the fault model we propose, with gate-level stuck-at fault coverage. The table shows that the application of the rules improves the predictive value of RT-level fault coverage figures.

	Correlation coefficient
Without rules	-0.1323
Rule A	0.6099
Rule A & B	0.7293
Rule A & B & C	0.7753

Table 5: Influence of the rules on correlation

Synthetic information is given in Table 5 and represented graphically in Figure 4 and Figure 5. The correlation coefficient between RT- and gate-level fault coverage figures is incredibly low if no rules are applied, thus exposing the difficulties of modeling at the RT-level faulty behaviors. However, as we apply the rules, we are able to reach a correlation coefficient around 77%. Rules A and C are more general, and give good results on all benchmarks. Rule B, on the other hand, contributes

significantly only for benchmarks that have poorly observable assignment statements, such as b04.

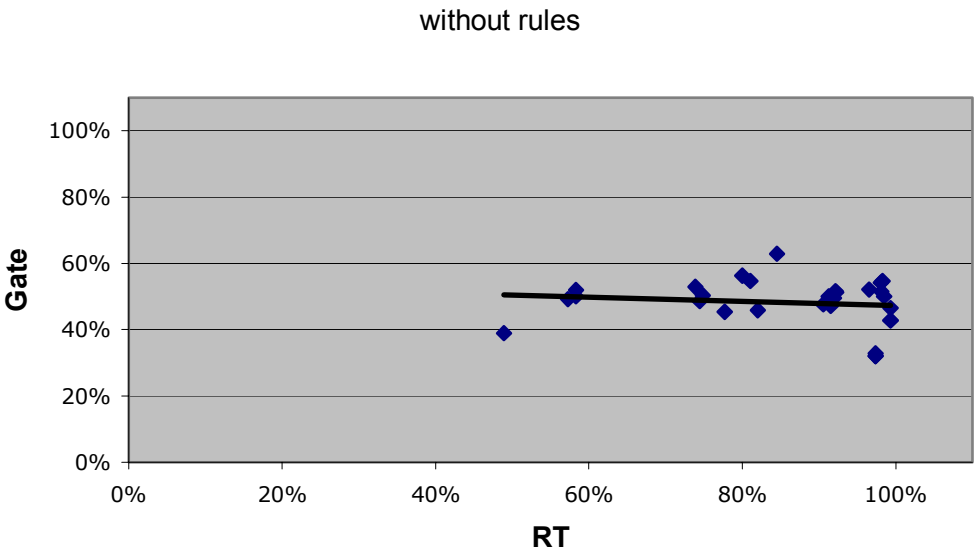

Figure 4: Correlation without rules

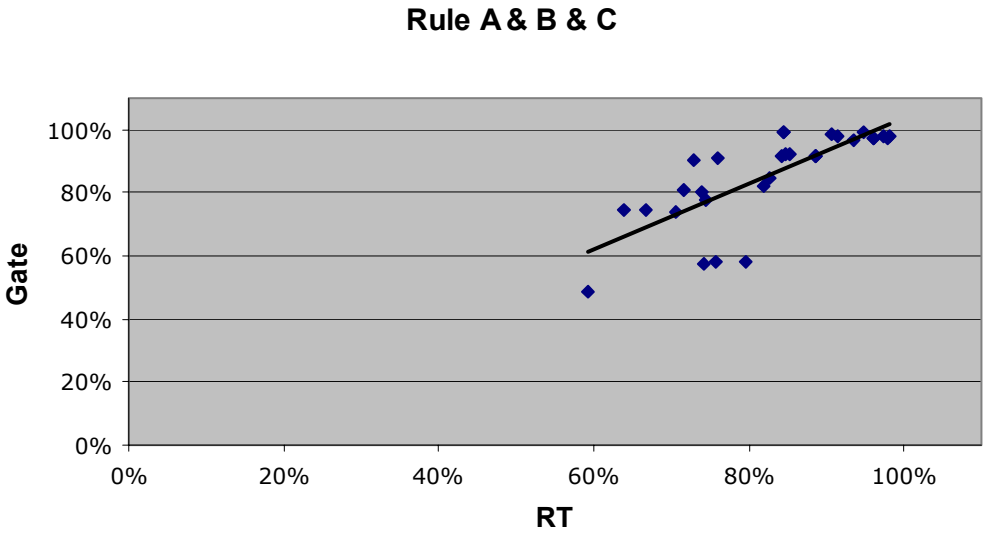

Figure 5: Correlation with rules

The experimental results show that the application of the redundancy identification rules allows the RT-level fault coverage to become more and more

correlated to the gate-level one. Circuits have an increment of fault coverage and an improvement of the global correlation value.

An exception is the circuit b07; in this case the presence of a ROM decreases the global correlation.

3 High-Level ATPG

High-level test pattern generation is increasing its industrial relevance [PITC99]. Designers would like to foresee an ASIC testability before starting its logic synthesis. The design practice is pushing the insertion of design for testability structures up to the RT-level, and their effectiveness should be evaluated as soon as possible. In addition, it has been increasingly observed that gate-level sequential automatic test pattern generation techniques may take unacceptable amounts of computing time and resources to tackle larger sequential circuits unless design-for-testability structures are used. High-level ATPG tools are expected to exploit compact information about design structure and behavior, and to generate high-quality test sequences more efficiently. Moreover, it is supposed that high-level generated test benches could be able to detect faults that would be very hard for gate-level ATPGs [SGTT00].

Tackling test issues above the gate-level is a hard task, and the lack of a fault model is one of the hardest theoretical barriers.

Code-coverage based fault models, deriving from the software testing field, may seem suitable to be applied on HDL descriptions. However, coverage metrics such as line/block coverage, branch/conditional coverage, expression coverage and path coverage lack of direct relationships with gate-level stuck-at faults, and their applicability in the field of test is difficult. Other considerable difficulties stem from the large amount of concurrency, from the complexity of timing schemes and from the combined presence of behavioral and structural description styles. But, definitely, the main problem with code-coverage based fault models is probably the lack of an explicit observability concept. Coverage metrics only consider *reachability*, that is

like *fault controllability* in the gate-level domain. However, any ATPG should tackle faulty-behavior *observation* as well [DGke96].

[RLJh98] presents TAO, a two-pass approach using a symbolic RTL test generator. The proposed testing paradigm involves writing path equations for modules, given the RTL connectivity, and solving them to obtain regular expressions for control paths.

Probably, the most successful proposal of a hardware-related high-level fault model is *Observability-Enhanced Statement Coverage* [FDKe98]. It introduces the concept of *tag* as the possibility that an incorrect value is computed at a given location, thus approximating the effects of fault propagation. Since this fault model does not assume any specific fault effect, its generality prevents explicit fault simulation.

The first ATPG exploiting *Observability-Enhanced Statement Coverage was* presented in [FADe99]. The vector generation procedure is based on hybrid linear programming and Boolean satisfiability methods.

ARTIST, a different RT-level ATPG exploiting high-level information to reach high code-coverage figures, was presented in [CSSq00]. Differently from [FADe99], ARTIST is a simulation-based approach. It is based on an evolutionary algorithm coupled with a commercial VHDL simulator, and due to the adoption of a commercial tool, it is able to produce sequences for general synthesizable VHDL description, with few limitations in complexity and characteristics, and it does not require any effort for re-modeling circuits or extracting special information. However, neglecting observability, sequences generated by ARTIST are not optimized for test purpose.

In [CSSq00a], ARTIST code-coverage metric was augmented with simplified observability. Fault-coverage figures dramatically increased, but the lack of a real fault model prevented the usage of a fault-dropping mechanism. ARTIST was given the goal to increase an observability measure, without meaningful stopping condition. Thus, the approach was not suitable for larger designs.

In [CCSS00] an extension of observability-enhanced statement coverage was proposed. In the new model, explicit *RT-level assignment single-bit* stuck-at's are used instead of generic tags. An RT-level assignment single-bit stuck-at fault is

defined as a single-bit stuck-at in the effect of an RT-level assignment operation: when a fault is present, the affected object (signal or variable target of an assignment statement) loads the correct value, except for one bit that is forced to 0 or 1. Experimental figures show that this model is highly correlated with gate-level coverage.

In [FFFFS01] Ferrara et al. presented BEHATE, an RT-level tool based on a metric called *bit-coverage*, close to the *RT-level assignment single-bit*. Although the paper is aimed at functional verification, experimental results show a strong relation between high- and gate-level faults.

[CCSS00a] shows a simulation techniques based on simulation command scripts that allows efficient exploitation of *RT-level assignment single-bit* faults. Using the Tcl interface of a commercial simulator, the simulation of each faulty circuit is shown no more costly than simulation of the original circuit.

This Chapter describes ARPIA, a high-level evolutionary automatic test signals generator. Experimental results gathered using the prototypical implementation is remarkable. The effectiveness of the generated test signals is at least comparable with (and in several cases higher than) that of previously proposed approaches. Additionally, thanks to the evolutionary algorithm and to a fault dropping mechanism, computational requirements of the new system are lower.

3.1 ARPIA

ARPIA is a simulation-based evolutionary test signals generator. Being an evolutionary algorithm, it evolves a population seeking fitter individuals. But, since individuals are test sequences for a digital circuit, the fitness measures the sequence ability to detect faults in the design. And it is computed by simulation. Given a fault model, the *fault coverage* is defined as the percentage of faults that the test sequence is able to detect. Thus, the goal of ARPIA can be rephrased as “generate a sequence of signals that attains maximum fault coverage.”

ARPIA shares the same philosophy with [CSSq00a]. They are both simulation-based approaches, and individuals are evaluated resorting to an RT-level fault simulator. However, the two methodologies exploit different fault models,

different fault simulation techniques and different evolutionary algorithms. Next Sections detail these three key points.

3.2 Fault Model

The RT-level single-bit stuck-at fault model was presented in [CCSS00] and described in Section 2.1. In this model, a fault is defined as a single-bit stuck-at in the effect of an RT-level assignment operation: when a fault is present, the affected object (signal or variable target of an assignment statement) loads the correct value, except for one bit that remains stuck to 0 or 1.

Faults are *single* and *permanent*: only one fault is inserted at a time and the fault effect is present during the whole simulation. The RT-Level single-bit stuck-at fault model does not explicitly consider control-flow faults, such as *stuck-at-true* or *stuck-at-false*.

Initially, the Fault List contains the list of all RT-Level single-bit stuck-at faults. However, during synthesis the RT level VHDL description is optimized in order to create an efficient gate-level design. The optimization process analyzes the VHDL description and simplifies all logic eliminating redundancies. In this phase some RT-level stuck-at faults lose their correspondent gate-level faults. In order to prevent this discrepancy is necessary to identify which parts of the logic described at the RT-level disappear during the optimization phase of the synthesis process and to eliminate the associated faults from the Fault List.

To perform Fault Simulation a serial fault simulation strategy is adopted. The good and each faulty machine are simulated, comparing their outputs. A fault is marked as *detected*, if it produces a difference on a Primary Outputs of the circuit at the end of a clock cycle. To run the simulations, the Test Pattern is first transformed to a set of commands that force the correct waveform for input signals, and the Fault List is transformed to a set of script commands for injecting faults during simulation.

Fault injection is made possible by creating routines that *change* the target signal/variable bit value during simulation, using the simulator scripting language (Tcl), when a given target assignment instruction is executed. The fault injection

procedures must face various issues derived from the fault model, from VHDL Semantics and from the simulator itself.

Further details can be found in Section 2.1.

3.3 Fault Simulation Technique

Fault simulation is made possible by creating routines that *change* the target signal/variable bit value during simulation, using the simulator scripting language (Tcl), when a given target assignment instruction is executed. The fault injection procedures, presented in [CCSS00a] and described in Section 2.2, must face various issues derived from the fault model, from VHDL semantics and from the simulator itself.

This fault simulation environment allows us to compute fault coverage figures at the RT-level with a minimal CPU time overhead, since the VHDL model of faulty circuits is simulated at the same speed as the fault-free model. The main time penalty is at fault activation time instants, where some breakpoints are set and TCL commands to modify values are executed.

3.4 Algorithm

A fault can be marked as *tested* only when it is both *excited* and *observed*. Given a fault, the target of the whole process is first to force the corresponding bit to a value that make the fault visible (excitation), then to propagate the fault effects to some primary output (observation).

The two problems are tackled separately, with two different strategies. Indeed, test generation is performed in three phases. The first is aimed at exciting faults, the second tackle observation, while the third dynamically optimize the fault list.

The goal of the first phase is to produce a set of signals able to excite an untested fault. The first phase implements a simple first-improvement hill climber (Figure 6). The local search procedure starts with a random sequence of given length L. In each step, a new sequence is generated by randomly mutating the current one. If the new sequence excites a larger number of untested faults, it becomes the current

one. Otherwise it is discarded. The process ends whenever the current sequence is able to excite at least *k* fault, or after a predefined amount of useless steps. It is worth noting that the number of excited faults is computed over the untested faults only.

```
current_sequence = random_sequence(L);
steady_state_factor = 0;
do
{
    new_sequence = mutate(current_sequence);
    if(excited_faults(new_sequence) >
excited_faults(current_sequence)) {
      current_sequence = new_sequence;
      steady_state_factor = 0;
    } else {
      increase(steady_state_factor);
    }
} while(excited_faults(current_sequence) == 0 &&
      steady_state_factor < steady_state_limit)
```

Figure 6: Phase One Pseudo Code

Three mutations are currently implemented by ARPIA: *add*, *delete* and *change*. The first two respectively add and delete a vector of input signals from the test sequence. The last one randomly changes a vector of input signals in the sequence.

When a test sequence able to excite a sufficient number of faults is found, it is transferred to the second phase together with the excited faults. The goal of the second phase is to observe each single fault in the excited set. This stage exploits an evolutionary algorithm similar to an evolution strategy [BHSc91].

First, a target fault f_t is selected from the excited set. Then, a population of P sequences is created by mutating the original sequence and evolved using a (P+P) strategy. In every evolution step, P new individuals are generated by mutating the P original ones (each sequence generates exactly one new sequence). The P fitter individuals are selected for survival among the 2P. The same three mutation operators of the first phase are adopted.

In the second phase, given a target fault f_t, the fitness measures how far a sequence is able to propagate f_t effects. More precisely, it is the maximum number of

differences caused by the fault during the application of a single vector of signals of the test sequence (1).

$$evaluation\_two(S, f_t) = \underset{v \in S}{MAX} \sum_{objects} different\_bits(object) \qquad (1)$$

The evolution is halted whenever f_t is detected or after a certain amount of generations. The pseudo code is shown in Figure 7.

The second phase is iterated until all faults in the excited set have been tested or aborted.

```
for t = 1 to P {
    population[t] = mutate(starting_sequence);
}
generations = 0;
success = 0;

do
{
    generations = generations + 1;
    for t = 1 to P {
      population[P + t] = mutate(population[t]);
      if(tested(ft, pupulation[t])
        success = 1;
    }
    sort(population);
} while(not success && generations < generations_limit);
```

Figure 7: Phase Two Pseudo Code

The evolution-strategy approach was chosen because of the complexity of the encoding. Individuals are sequence of vectors. Each vector of signals is simulated in a clock cycle. Circuits contain memory elements, thus the behavior in a clock cycle depends both on current input signals and previous ones. The effect of a traditional recombination operator, like the uniform crossover, can be very similar to a complete random mutation at phenotypic level. We shun any risk and exploit an algorithm that "omits recombination since its philosophy relies on species as evolving entities" [ScKu98].

After each successful second phase, an optimization mechanism called *fault dropping* is activated. All still untested faults are simulated with the new sequence,

seeking if any additional fault is detected by it. This is more of a possibility than an expectation, since the starting sequence found in the first phase is required to be able to excite more than one fault. The fault dropping mechanism greatly enhances overall algorithm performance.

3.5 Experimental Evaluation

In order to practically evaluate the effectiveness of the proposed approach, we implemented a prototype. The generator is composed of about 1,500 lines in ANSI C and interacts with V-System 5.3 VHDL simulator by Model Technology. Special techniques are adopted to speed-up fault simulation [CCSS00a]. During experiments we adopted the following parameter values:

- first-phase initial sequence length of 50 clock cycles ($L = 50$);
- first phase sequence required to excite 5 different faults ($k = 5$);
- second phase population of 10 individuals ($P = 10$).

Circuit	CPU time [s]	RT-level Faults			Gate-level Faults		
		Tot	Det	FC%	Tot	Det	FC%
b01	78,80	81	81	100,00%	258	258	100,00%
b02	39,19	43	39	90,70%	150	149	99,33%
b03	1.089,96	213	145	68,08%	822	615	74,82%
b04	1.627,86	424	353	83,25%	3.356	3.035	90,44%
b05	1.932,27	778	244	31,36%	5.552	1.856	33,43%
b06	200,31	110	82	74,55%	5.552	5.387	97,02%
b07	9.297,26	289	146	50,52%	2.404	1.401	58,28%
b08	2.832,38	154	118	76,62%	918	839	91,39%
b09	4.970,53	240	196	81,67%	900	768	85,33%
b10	778,35	172	127	73,84%	1.054	961	91,18%
b11	34.837,11	381	263	69,03%	2.868	2.614	91,14%
b12	7.890,20	870	115	13,22%	5.280	1.105	20,92%
b13	2.801,85	284	224	78,87%	1.818	1.501	82,56%
b14	473.741,90	10.493	9.114	86,86%	28.990	23.708	81,78%
b15	590.611,31	4.900	2.026	41,35%	55.568	18.060	32,50%

Table 6: ARPIA Result

Table 6 reports the experiments performed on the ITC99 RT-level benchmarks. These benchmarks are representative of typical circuits, or circuit parts, that can be

automatically synthesized as a whole with current tools and are described in [CSSq00]. Experiments have been run on a Sun Enterprise 250 running at 400 MHz and equipped with 2 Gbytes of RAM.

The first column of Table 6 reports the name of the benchmark, while the CPU time required to generate the test signal sequence is shown in the second column. RT-level fault figures are reported in the next column block in terms of: total number of faults [Tot], number of detected faults [Det] and percent fault coverage [FC%]. The next column block shows gate-level figures: total number of gate-level faults [Tot], number of detected [Det] and respective fault coverage [FC%].

Results show that ARPIA is able to generate test sequences that are highly effective both at RT-level and at gate-level, within an acceptable CPU time. However, to better analyze the tool performance, we need to compare it with different approach (Table 7).

Circuit	ARPIA (no ES)		ARTIST		ARPIA	
	RT	Gate	RT	Gate	RT	Gate
b01	100,00%	99,61%	100,00%	100,00%	100,00%	100,00%
b02	90,70%	99,33%	90,70%	99,33%	90,70%	99,33%
b03	56,81%	69,10%	68,08%	74,82%	68,08%	74,82%
b04	69,34%	69,19%	83,79%	91,03%	83,25%	90,44%
b05	11,31%	5,42%	31,36%	33,50%	31,36%	33,43%
b06	70,00%	93,38%	74,80%	97,35%	74,55%	97,02%
b07	47,75%	56,49%	50,52%	58,28%	50,52%	58,28%
b08	52,60%	28,00%	60,10%	71,68%	76,62%	91,39%
b09	62,50%	48,89%	77,84%	81,33%	81,67%	85,33%
b10	46,51%	65,84%	73,69%	90,99%	73,84%	91,18%
b11	43,57%	58,79%	68,64%	90,62%	69,03%	91,14%
b12	2,76%	4,36%	29,06%	45,99%	13,22%	20,92%
b13	31,69%	31,19%	n/a	n/a	78,87%	82,56%
b14	11,57%	37,91%	n/a	n/a	86,86%	81,78%
b15	14,69%	12,75%	n/a	n/a	41,35%	32,50%

Table 7: Comparison with ARTIST and ARPIA without Evolutionary Algorithm

The first column block of Table 7 reports data for the first prototype of ARPIA, where a simple hill-climber was exploited instead of the evolution strategy. The gap between the two RT-level figures shows the fundamental role played by the

evolutionary algorithm. The gap between gate-level fault coverage statistics is a mere consequence.

In the second column group of Table 7 we reported results attained by ARTIST [CSSq00a], a highly-optimized tool exploiting a genetic algorithm. The difference between these two tools can be explained resorting to both the fault model and the new evolutionary mechanism. A deeper comparison between the results of ARPIA and ARTIST (complete data can not be reported here for lack of space) shows that the former is characterized by a higher efficiency (i.e., it requires a lower CPU time), thanks to fault dropping and a higher compactness of the generated sequences. Indeed, ARTIST was not able to tackle some of the benchmarks due its lower efficiency (marked with "n/a" in the table).

Circuit	RT-Level Faults				Phase2 Efficacy
	Tot	Exc	Det	Err	
b01	81	81	81	0	100,00%
b02	43	43	39	4	100,00%
b03	213	148	145	3	100,00%
b04	424	356	353	3	100,00%
b05	778	340	244	19	76,01%
b06	110	87	82	5	100,00%
b07	289	240	146	20	66,36%
b08	154	118	118	0	100,00%
b09	240	196	196	0	100,00%
b10	172	139	127	12	100,00%
b11	381	289	263	26	100,00%
b12	870	130	115	1	89,15%
b13	284	266	224	16	89,60%
b14	10.493	10.165	9.114	0	89,66%
b15	4.900	2.244	2.026	24	91,26%

Table 8: Phase Two Effectiveness

The effectiveness of the evolutionary algorithm can also be seen in Table 8, where the number of excited faults is shown in column [Exc] together with the number of detected faults [Det]. The effectiveness of the evolutionary algorithm can be defined as its ability to observe (i.e., to detect) excited faults, not considering faults that are certainly unobservable due to incorrect design ([Err] column).

It should be noted that phase two effectiveness is quite high also for b12, a problematic circuit where ARPIA only manages to get 13.22% RT-level fault coverage. Thus, primarily the first phase can be hold responsible for the low performance.

4 Microprocessor Test

Microprocessors and microcontrollers are known to be major challenges in the test arena, due to their complexity and heterogeneity. Techniques for microprocessor testing can be first divided in two groups, depending on whether implementation information are available (for microprocessor producers) or not (when users adopt producer-independent incoming inspection test). In the latter case, only high-level functional information are available, and test solutions can not rely on any knowledge about the real implementation of the device. A similar situation arises when soft IP cores are designed, and suitable input sequences are required, able to test them no matter the technology re-mapping and the environment the core is embedded in.

In both the above cases, any Design for Testability technique can hardly be considered, and an effective solution is to devise a test program to be executed by the microprocessor itself. Its behavior must be monitored, and possible mismatches signal the existence of one or more faults inside the processor.

Traditionally, the test of a microprocessor has been performed by resorting to functional approaches based on exciting all the functions and resources described in its data-sheets [ThAb80]. This approach involves a high amount of manual work performed by skilled programmers, and does not provide any quantitative measure about the attained Fault Coverage (FC). Recently, Dey et al. proposed a deterministic method named DEFUSE [ChDe00] to generate test programs able to reach a good Fault Coverage on the ALU of a microprocessor, and to compact the result. The approach is very effective with combinationally testable parts (e.g., simple ALUs), but shows some limitation when hard-to-test sequential modules, such as Control Units, are addressed. Another approach has been proposed by Batcher and Papachristou [BaPa99] that is based on generating random sequences of instructions, but it requires

the insertion of additional hardware in the microprocessor under test. Recently, Sheen et al. proposed a technique where the processor itself generates test at run-time by self-modifying code [PMNo99]. On the other hand, Utamaphethai et al. showed a method for generating instruction sequences for validating the branch prediction mechanism of the PowerPC604 [NCPa92]. Generated sequences are very effective, but the methodology exploits a deep knowledge of the processor and is not straightforward to be applied on general designs.

The methodologies for developing test programs can be divided in three types: *manual*, *semi-automatic* and *automatic*. The first one relies on internal knowledge of processor and requires skilled programmer, expert in architecture and test. The semi-automatic methodology is based on the idea of let an ATPG assembles and refines some manually prepared code fragments. In the last one an ATPG generates automatically test programs on the base of an instructions set description.

In the next sections a semi-automatic and an automatic method for developing Test programs are described analyzing their advantages and limitations.

4.1 A Semi-Automatic Test Program Generation Methodology

The method partly stems from the ideas already introduced in [CSSq01], but thanks to the adoption of an effective RT-level fault model [CCSS00], any reference to the gate-level netlist is avoided. The proposed method requires a limited amount of manual work aimed at developing a library of *macros*, which are able to excite all the functions of the processor and to make the effects of possible faults observable. A macro is required for every machine-level instruction; each macro is composed of few instructions, aimed at activating the target instruction with some generic operand values representing the macro parameters, and to propagate to an observable memory position the results of its execution. The complexity of the work for developing these macros and the required skills are much lower than for the approaches based on functional testing, such as [ThAb80]; in fact, our approach just requires the development of one macro for every machine-level instruction according to a simple pre-defined skeleton for every group of instructions, and does not involve the

extraction of complex graphs describing the relationships among resources, as in [ThAb80]. The final test program is composed of a proper sequence of macros taken from this library, each activated with proper values for its parameters (i.e., the operands of the composing instructions). This phase is accomplished by resorting to a Genetic Algorithm which exploits an RT-level Fault Simulator to evaluate the generated solutions. Experimental results supporting the effectiveness of the method are reported for a core of the Intel 8051 microcontroller using a prototypical implementation of our algorithm. A synthesizable VHDL RT-level description of the microprocessor is used. Final figures show that the test program generated by the tool has a higher effectiveness (in terms of attained gate-level fault coverage) than the one generated by the gate-level test program generation method introduced in [CSSq01], and that the required computational effort is comparable between the two approaches.

4.1.1 Test Strategy

Test sequence generation for microprocessors necessarily requires the knowledge of the processor instruction set and instruction format, since only correct programs can internally perform meaningful operations. A solution for this problem was proposed in [CSS00] with the usage of *macros*: a short sequence of instructions aiming at creating a suitable framework for testing the part of control unit and data-path affected by a given instruction (or group of instructions).

The purpose of macros is to execute all the possible instructions and to make observable the complete result of each instruction, which also includes any flag that is possibly affected by the instruction itself.

```
MOV AX, K1  ;load register AX with K1
MOV BX, K2  ;load register BX with K2
ADD AX, BX  ;sum BX to AX
MOV RW, AX  ;write AX to RW
MOV RW2, PSW  ;write status register to RW
```

Figure 8: Pseudo-code of the macro for the ADD instruction.

As an example, Figure 8 reports the code (for sake of readability we use a pseudo-assembly language) for the macro concerning the addition instruction between registers using `K1` and `K2` as parameters. RW and RW2 are two easily observable memory locations.

Macros are stored in a library. A test program is a collection of macros. An optimization algorithm selects the most suitable ones from the library, and defines the values of their parameters.

The effectiveness of test program generation for microprocessors RT-level descriptions strongly depends on the adopted RT-level fault model. We selected the *RT-Level single-bit stuck-at* fault model [CCSS00] that shows a good correlation with gate-level stuck-at faults.

An RT-level single-bit stuck-at fault is defined as a single-bit stuck-at in the effect of an RT-level assignment operation: when a fault is present, the affected object (signal or variable target of an assignment statement) loads the correct value, except for one bit that remains stuck to 0 or 1. The effect VHDL statement is the statement *corresponding* to the fault. The faults are *single* and *permanent*: only one fault is inserted at a time and the fault effect is present during the whole simulation. Other assignments of the same signal are assumed to be fault-free, since stuck-at faults on the same signal but on different statements are considered different.

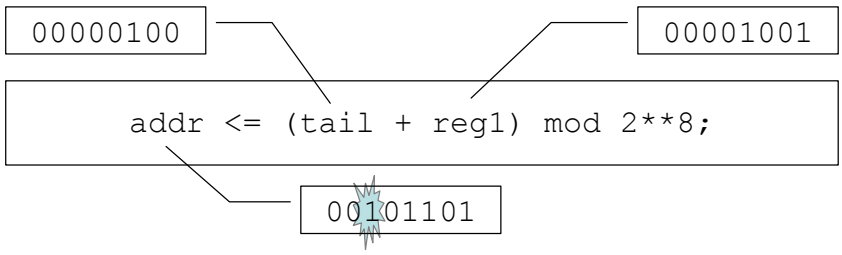

Figure 9: RT-level Single bit Stuck-at Fault Example.

Figure 9 shows the example of a RT-level single bit stuck-at fault. The fault affects the third bit of the assignment operation, and modifies the result of the expression, after it has been computed and before it is assigned to the target signal.

Better correlation with the gate-level fault model is obtained with the application of some *Fault Collapsing rules*, described in Section 2.1.2, able to partially eliminate the RT-level faults that do not correspond to any gate-level fault after synthesis.

4.1.2 Test Program Generation

To perform Test Program Generation starting from the analysis of the VHDL description, we must select the best macros and the values of their parameters in order to create a program able to detect the highest number of faults.

The environment we propose, whose architecture is shown in Figure 10, is composed of:

- *Fault Manager,* that analyzes the VHDL description and creates a Fault List, according to the *RT-Level single-bit stuck-at* fault model and using the *Fault Collapsing* rules introduced above;
- a *Core* that, using a set of heuristics (i.e., greedy, hill climber and evolutionary algorithms), selects the most suitable macros and the values for their parameters to create the test program;
- a *Fault Injector* that, interacting with the *Fault Simulator*, injects the faults on the microprocessor RT-level description and evaluates the effectiveness of the macros created by the *Core*.

After generating the Fault List, the faults are injected during the simulation whenever the corresponding statement is executed. All the faults corresponding to a statement which has been executed at least once by the test program are labeled as *executed*. Once a fault is executed, it is also excited, if the corresponding bit assumes a value in the fault-free system which is the opposite of the stuck-at one. Finally, when a fault produces at least one difference in the output behavior of the processor (in terms of produced and observable results) it is marked as *detected*.

As we work on a microcontroller description, we can group faults in two classes:

- detectable independently from macro operands;
- detectable only using a specific set of macro operands.

In this work, the faults that belong to the first class are called *control-dependent* faults and the ones belonging to the second class are called *data-dependent* faults. Based on our experience most of the *control-dependent* faults are located in the Control Unit and in the Instruction Decoder, where the systems decides *how* to elaborate the instruction data. Instead, most of the *data-dependent* faults are located in the data path (e.g., in the Arithmetic and Logical Unit).

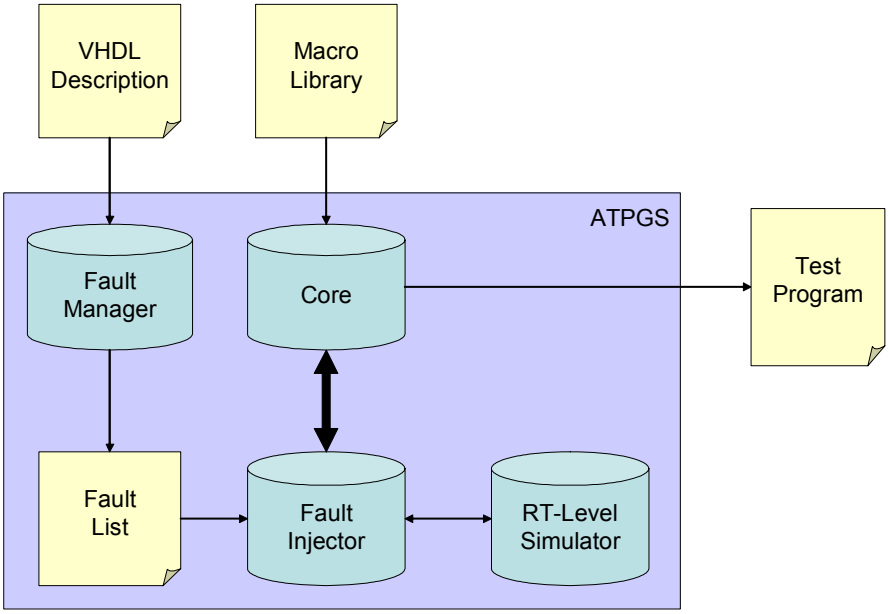

Figure 10: Test Program Generation Architecture.

4.1.3 Algorithm

The algorithm we propose is based on two phases:

- *control-dependent* fault detection phase;
- *data-dependent* fault detection phase.

The detection of *control-dependent* faults is based on the correct selection of the operative code and the addressing mode. The detection of these faults depends on which instructions (i.e., which macros) have been executed by the microprocessor, independently from a specific set of data to be used as macro parameters. For this reason, a first phase is activated, which aims at maximizing the number of detected *control-depending* faults. The pseudo-code of this phase is reported in Figure 11.

At each iteration, the procedure `select_best_macro` simulates each macro in the library with random operands. By means of this procedure the VHDL statements executed during the fault-free simulation of each macro are identified. The macro **M** that maximizes the number of executed faults is selected. The selected macro is then fault simulated and, if at least one new fault is detected, it is added to the final test program. The macro **M** is also marked as used to avoid being selected again in this phase.

```
do
{    (M,O) = select_best_macro();
     F = compute_detected_faults(M,O);
     if( F is not empty )
       add M(O) to the test program;
     drop_faults(F);
     remove M from selectable_macros;
} while(stopping_condition is false)
```

Figure 11: Control-dependent fault detection phase pseudo-code.

When all the macros of the library have been selected and fault simulated, all the macros become selectable again and the second phase starts.

The goal of the second phase is to detect *data-dependent* faults. The coverage of these faults depends on the arguments of each instruction (i.e., macro operands) executed by the microprocessor. The pseudo-code of this phase is reported in Figure 12.

As in the first phase, at each iteration the instructions executed by each macro of the library are first identified via fault-free simulation. The macro **M**, that maximizes the number of executed faults is selected.

A *hill-climbing* algorithm is then activated, whose goal is to find the values for the macro operands (**O**) that maximize the number of faults excited by the macro. At the beginning a set of random operands $\mathbf{O_{max}}$ is created and the number of faults $\mathbf{N_{max}}$ activated by the macro $\mathbf{M(O_{max})}$ is computed. At each iteration a new set of operands $\mathbf{O_{new}}$ is created applying local transformations (i.e., changing some bit values) to $\mathbf{O_{max}}$. If the number of faults $\mathbf{N_{new}}$ activated by the macro $\mathbf{M(O_{new})}$ is higher than $\mathbf{N_{max}}$, $\mathbf{O_{max}}$

and $\mathbf{N_{max}}$ are substituted by $\mathbf{O_{new}}$ and $\mathbf{N_{new}}$, and a new iteration starts. The *hill-climber* runs until the number of activated faults reaches a given threshold, or the maximum number of iterations has been reached.

```
do
{   M = select_best_macro();
    do
    { O = select_operands(M);  /*hill climber*/
      F = compute_detected_faults(M, O);
      if( F is not empty )
      { add M(O) to the test program;
        drop_faults(F); }
        A = compute_activated_faults(M, O);
        do
        { Ft = select_fault(A);
          O = optimize_the_operands(M, Ft);
          if( Ft is detected )
          { add M(O) to the test program;
            fault_dropping(M, 0); }
        }while( A is not empty );
    }while( stopping_condition() == FALSE );
    remove M from selectable_macro;
}while( selectable_macro is not empty );
```

Figure 12: Data-dependent fault detection phase pseudo-code.

For each fault $\mathbf{F_t}$ activated by the selected macro, a Genetic Algorithm, detailed in next Section, is then executed, whose goal is to find the values for the macro operands ($\mathbf{O}$) that detect the target fault.

If $\mathbf{F_t}$ is detected, the macro is added to the final test program and a fault dropping phase is activated; otherwise, the fault is discarded, to avoid being considered again with this macro.

When all the activated faults have been detected or discarded, the algorithm returns to the *hill-climber* in order to try to activate others faults.

The stopping condition is true when either the Fault Coverage reaches a given threshold, or the maximum number of iterations has been reached.

When the stopping condition is reached, the selected macro is marked as used, all the faults discarded are reinserted in the Fault List, and a new iteration starts.

The *data-dependent* faults detection phase ends when either the Fault Coverage reaches a given threshold, or all the macros of the library have been selected.

4.1.4 Genetic Algorithm

Once a macro has been selected from the library, a fault simulation is performed. For each fault excited by the selected macro, a Genetic Algorithm (GA) is then activated.

The goal of the GA is to identify the best values for the parameters of the selected macro in order to detect the target fault. The algorithm chooses the values for immediate operands, and those to be written in the registers or memory cells used by the target instruction.

The number of operands and their length (in bits) depend on the macro. A standard steady-state Genetic Algorithm is exploited, whose main characteristics are summarized in the following:

- chromosomes are bit strings corresponding to the concatenated operands; their length is function of the macro;
- the mutation operator randomly selects a bit in the chromosome, and complements it;
- the cross-over operator is the standard one-cut crossover;
- chromosomes are selected using a linearized fitness function and a roulette wheel mechanism.

The fitness function of a chromosome measures how far the macro **M**, created with the chromosome parameters **O**, is able to propagate the target fault $\mathbf{F_t}$ effects. More precisely, it is the maximum number of differences caused by the fault during the execution of the macro.

$$Fitness(M, O, F_t) = \underset{v \in S}{MAX} \sum_{objects} different\_bits(object)$$

where *different_bits* counts the number of bits having a different value in the fault-free and faulty system for any VHDL objects (i.e., signal and variable). The fitness function calculates the sum of differences at every clock cycle of any execution of the macro and takes the maximum.

The algorithm is stopped when the target fault is detected or a steady state is reached, i.e., when a given number of generations have elapsed without detecting the target fault.

4.1.5 Experimental Evaluations

In order to test the effectiveness of this semi-automatic technique we implemented it in a tool called *Automatic Test Program Generation System* (ATPGS). ATPGS amounts to about 11,000 lines of C code including an in-house developed RT-Level Fault Simulator based on a commercial VHDL Simulator (ModelSim 5.5a by Mentor Graphics).

The system has been evaluated on a synthesizable VHDL description of the Intel 8051 microcontroller, containing the core system without peripherals, whose main characteristics are summarized in Table 9.

Primary inputs	41
Primary outputs	45
VHDL lines	13,583
Processes	6
Procedures	29
RT-level faults	15,387
Gates	12,134
Flip flops	1,325
Gate-level faults	28,792

Table 9: 8051 description characteristics.

The Fault Simulator is able to simulate the entire 8051 while it executes the program stored in the embedded ROM, injecting RT-level single bit stuck-at faults in VHDL code. ATPGS is able to modify the program stored in the 8051 ROM without recompiling the entire VHDL code but interacting with the simulator in order to modify at runtime the map of the ROM. A library of 115 macros is exploited, each composed of a number of instructions that ranges from 3 to 6.

The experiments have been performed on a Sun Enterprise 250 running at 400 MHz and equipped with 2 GBytes of RAM.

Parameter	Value
Number of individuals in the population	25
Number of new individuals at each generation	25
Maximum number of generations without improvements	10
Crossover probability	0.7
Mutation probability	0.3

Table 10: Genetic Algorithm Parameters.

The values we used for the Genetic Algorithm parameters are reported in Table 10.

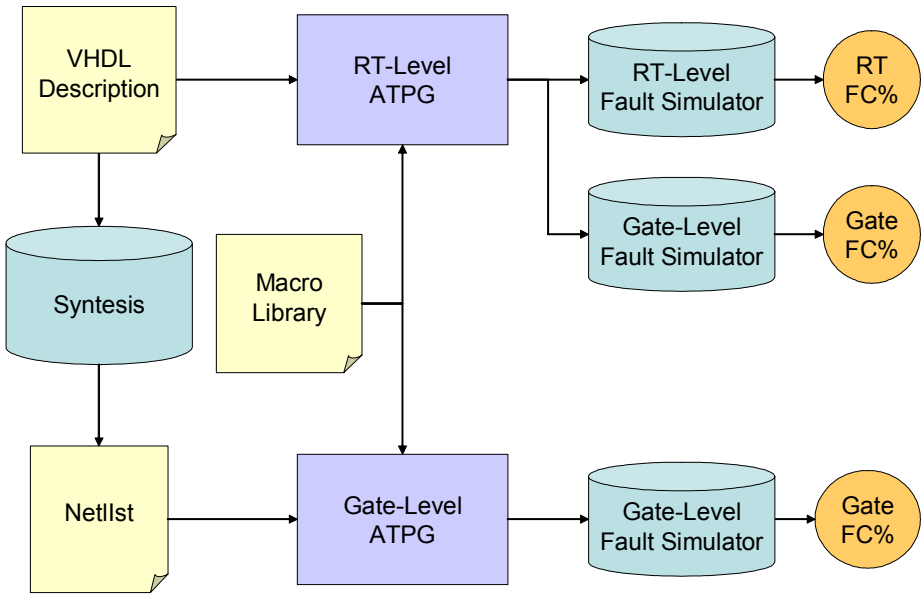

Figure 13: Experimental setup for comparison purposes.

For the purpose of the experiments, the RT-level ATPG was first run, with the goal of maximizing the Fault Coverage based on the RT-level fault model and a test program was obtained. For comparison purposes, a second set of experiments was then performed: the RT-level description of the 8051 was synthesized and fault simulated at gate-level. Using this description, we compared the results obtained by the RT-level ATPG (Table 11) with the Fault Coverage obtained by the gate-level

ATPG described in [CSSq01] (Table 12). The whole procedure adopted for the experiments is outlined in Figure 13.

The reported results show the proposed technique provides Fault Coverage figures higher than the gate-level ones, with a slight increase in the length of the final test program (in terms of number of instructions).

The RT-level ATPG works on a Fault List created analyzing the VHDL description and reduced by the Fault Manager applying the Fault Collapsing rules.

RT-level faults [#]	15,387
Executed faults [#]	13,364
Excited faults [#]	12,263
Detected faults [#]	12,122
Test Program Instructions [#]	883
RT-level Fault Coverage [%]	78.78
Gate-level Fault Coverage [%]	89.47

Table 11: Test Program generation from RT-level description.

Gate-level faults [#]	28,792
Detected faults [#]	25,759
Test Program Instructions [#]	624
Gate-level Fault Coverage [%]	85.19

Table 12: Test Program generation from gate-level description.

In the 8051 Fault List above 40% of the faults are eliminated by the usage of the rules; this happens because many internal parameters, especially in the Instruction Decoder and in the Control Unit, are constants. Before the proposed method is evaluated in terms of required computational effort, it must be first emphasized that in the current implementation of the tool the RT-level Fault Simulation is performed exploiting a commercial VHDL simulator. The interaction with it is necessarily loose, and therefore slow. However, if the method were integrated in the code of the simulator, a much higher efficiency would be attained. For this reason, to evaluate the required computational effort, we adopted as a parameter the number of 8051

instructions simulated by ATPGS during the test program generation phase. This number is equal to about two million instructions, and roughly corresponds to the number of instructions simulated by the gate-level ATPG described in [CSSq01].

4.1.6 Methodology Limits

The semi-automatic methodology described in the previous paragraph is not suitable for advanced processor architectures that contain parts (e.g., pipelines and caches) whose behavior is determined by a sequence of instructions and by the interaction between their operands.

Pipelined microprocessors, in particular, are complex and critical designs. A pipeline contains several independent units, called *stages*, and each stage executes concurrently, feeding its results to following units. Instruction execution-steps are arranged so that the CPU does not have to wait for one operation to finish before starting the next: consecutive instructions are likely to have their execution overlapped in time.

The first interesting consequence is that the behavior of a pipeline is not determined by *one* instruction and its operands, but by a *sequence* of instructions and all their operands. The simultaneous execution of multiple instructions leads to additional difficulties. Reviewing all pipeline peculiarities would deserve a very long discussion, but it may be insightful sketching three types of potential problems: *data*, *control* and *structural* hazards.

Data hazards are caused by data dependency between instructions: for instance, one instruction may depend on the result of a previous one, already in the pipeline. They are usually classified as *RAW* (read after write), *WAR* (write after read) and *WAW* (write after write) depending on the operations involved. Control hazards are caused by instructions that alter the usual flow of the program. For example, a conditional branching instruction invalidates the execution of all instructions following the incorrectly-predicted branch. Finally, structural hazards are produced by instructions contenting non-sharable resources, such as the floating-point unit.

In all cases, the simplest solution is to “stall” the pipeline until the hazard is resolved, however an excessive stalling may significantly degrade the overall

performance. Thus, to reduce stalls, designers adopt mechanisms, such as *data forwarding*. The basic idea of data forwarding is to pass a result directly to the functional unit that needs it, forwarding data from the output of one unit to the input of functional unit(s) requiring it.

Test programs may be exploited for verification, during design process, or for post-production test. Simulating the execution of appropriate test programs is a standard step in any verification process, even when formal techniques or other advanced methodologies are exploited. Furthermore, a test strategy based on a test program is broadly applicable. It does not rely on the insertion of special test architectures such as scan chains, thus it may be the only viable solution for microprocessors embedded inside a *SOC* (System-On-a-Chip). Furthermore, it allows an at-speed testing, an essential attribute for testing delay faults.

Despite its potential usefulness, automatically devising test programs for pipelined microprocessors is still an open problem (which becomes even bigger when *superscalar* architectures are considered, where two or more operations are executed in parallel). As mentioned before, it is not sufficient to check the functionalities of all possible instructions with all possible operands, but it is necessary to check all possible interactions between instructions and operands inside the pipeline. The task is not trivial, as data forwarding and similar mechanisms may lead to complex interactions.

In the literature, microprocessor test has been traditionally performed resorting to functional approaches based on exciting functions and resources ([ThAb80], [ShSu88]). However, these methodologies involve high amount of manual work.

More recently, [ShAb98] proposes a methodology to synthesize a self-test program not based on an a-priori fault model. The approach generates a sequence of instructions that enumerates all the combination of the operations and systematically selects operands. However, users need to determine the heuristics to assign values to instruction operands to achieve high stuck-at fault coverage. In some cases, this might not be a trivial task.

[KPGZ02] describes a methodology that allows devising an effective test program for a microprocessor core. However, the method requires that test engineers

create deterministic test patterns to excite the entire set of operations performed by each component of the core.

Unfortunately, all these approaches disregard pipeline behavior, or implicitly assume all single instructions to be independent, leading to a questionable efficacy.

Other techniques are potentially able to tackle pipelined designs. [BaPa99], for instance, is based on generating random sequences of instructions and is able to attain a fairly high level of fault coverage on non-pipelined architecture. However it requires the insertion of BIST circuitry, and assumes that all instructions are single-cycle and busses are never floating.

[LeSi91] proposes an interesting approach for functional testing of pipelined processors, but generated tests were extremely large. [SATa96] presents a more effective methodology. However, both methodologies require high amounts of manual work.

Since verification and test have several common points, papers dealing with pipelined microprocessor verification are also of deep interest. [UBSh99] showed a method for generating instruction sequences for validating the branch prediction mechanism of the PowerPC604. Generated sequences are very effective, but the methodology exploits a deep knowledge of the target processor and cannot be easily applied on general designs.

[HeDu01] showed how a pipelined processor can be modeled with a high-level behavioral HDL description. Authors manage to fit the model in a moderately sized FPGA and exploited it for various analysis, included testability. The approach requires high amount of manual work performed by skilled experts, and its efficacy depends on how features are captured by the model and may be easily biased by engineers' opinions.

4.2 An Automatic Test Program Generation Methodology

The approach presented in these paragraphs, differently from the above, is automatic, broadly-applicable and does not rely on skilled experts. It exploits an evolutionary technique called *genetic programming* ([Koza98]) and automatically *induces* assembly programs for maximizing a defined metric. Test-program generator

parameters are *auto-adapted* to their optimal values automatically and human intervention is limited to the enumeration of all available instructions and their possible operands. The methodology is able to tackle both conventional and pipelined processors, and can be utilized both for test and verification.

In modern μPs instructions are *pipelined*; this means that consecutively-executing instructions can have their execution overlapped in time. The details of instruction execution are arranged so that the CPU doesn't have to wait for one operation to finish before starting the next.

```
            RegA = 100;
            GOTO LABEL;
            RegA = 0;
            RegA = 10;
LABEL:
            RegA = RegA + 1;
```

Figure 14: Pipeline Effects

A striking peculiarity concerns program branches. When the CPU is ready to execute an instruction, it must first *fetch* that instruction (asking memory to retrieve the instruction at the appropriate address) and then *execute* that instruction (figuring out which operation is specified by that instruction and actually carrying it out). In modern pipelined architectures, at any given time, the CPU will be executing some instructions and, *at the same time*, it will be fetching the next instructions in the program. When a jump instruction is executed (for example, a call to a subroutine), the instruction appearing immediately after the call or jump in the code is already fetched in the pipeline. Thus, depending on μP architecture and implementation, the instruction following a branch may execute regardless of which way the branch goes.

For instance, after the pseudo-code in Figure 14, variable A holds 1 and not 101. Several hazards also arise from data dependencies, when consecutive instructions operate on the same data (in Figure 14, for instance, consecutive instructions read and modify the value of register *RegA*, resulting in a hazard).

Pipelined-processor assembly-language programmer must be constantly aware of all these problems while coding. Fortunately, modern compilers handle these

peculiarities automatically and most of high-level programmers may ignore them; however, a test program able to test the pipeline must be written directly in machine code.

4.2.1 Test-program Generation

The overall architecture of the proposed approach is shown in Figure 15.

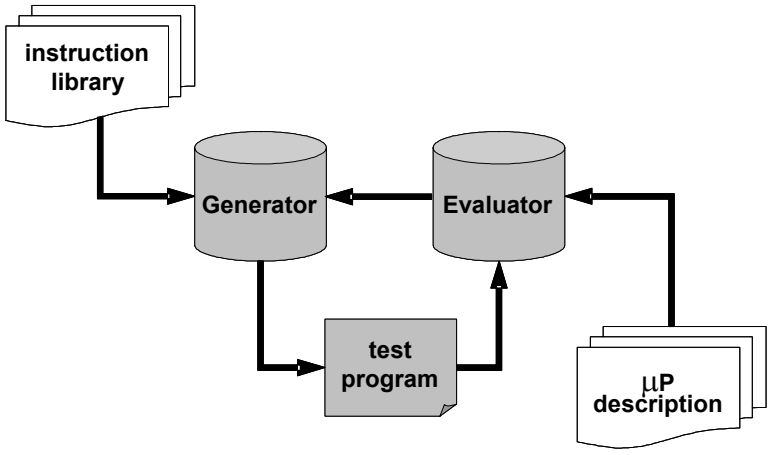

Figure 15: System Architecture

The *Generator* cultivates a population of test programs, exploiting the description of the syntax of the microprocessor assembly language stored in an external *instruction library*. Induced test programs are evaluated by an external *Evaluator* that provides feedback to the *Generator*.

In particular, the method demonstrates to efficiently generate test programs for the two microprocessors, able to maximize the value of two target metrics, i.e., the statement coverage on RT-level descriptions, and the toggle activity on gate-level ones.

Next Sections better detail the approach.

4.2.2 Program Evaluation

Test-program evaluation associates a fitness value to each test program. This value is used to probabilistically select the λ parents for generating new offspring and

to deterministically select the best μ individuals surviving at the end of each evolution step.

Different evaluation functions are used accordingly to the goal of the test-program generation. The loose coupling between the test-program generator and the test-program evaluator extends the applicability of the approach.

In this work, two different evaluation functions are exploited: *RT-level statement coverage* and *gate-level toggle activity*.

The former is one of the simplest verification metrics and can be considered as a required starting point for any simulation-based design-verification process. Statement coverage analysis ensures that no part of the design missed functional test during simulation, as well as reducing simulation effort from "over-verification" or redundant testing. Moreover, use of coverage analysis provides an easy and objective, although insufficient, way of measuring simulation effectiveness to ensure that all bugs would be exposed with the minimum amount of effort. Indeed, most CAD vendors have recently added code-coverage features to their simulators.

The *toggle activity* metric is a well-known metric for measuring functional vectors at the gate-level and for validating the integrity of a test suite [KaNo96]. It measures the percentage of *toggled* nodes, i.e., the percentage of nodes that switch from logic 0 to logic 1 and vice-versa. It was adopted as a preliminary step towards the generation of programs able to test gate-level stuck-at faults: being able to toggle all nodes is equivalent of being able to *excite* gate-level stuck-at faults; obviously observability is not taken into account.

4.2.3 Assembly-Level Test-Program Induction

Genetic Programming (GP) was defined as a domain-independent problem-solving approach in which computer programs are evolved to solve, or approximately solve, problems [Koza98]. GP addresses one of the more desired goals of computer science: creating, in an automated way, computer programs able to solve problems.

In GP context programs are usually represented as *tree*. A tree is a special kind of directed acyclic graph where there is only one path between any two nodes. Tree representations have been traditionally implemented in the *LISP* language as

S-expressions. However, in recent year, several researchers proposed to modify this conventional representation.

Remarkably, in [Hand94] the whole population was stored as a single directed acyclic graph, rather than as a forest of trees, leading to considerable savings of memory (structurally identical sub-trees are not duplicated) and computation (the value computed by each sub-tree for each fitness case can be cached). In [Poli97] a significant speed-up was achieved extending the representation from trees to generic graphs and parallelizing the evolution process.

For the purpose of this works, however, it is more interesting to examine techniques based on the idea of *compiling* GP programs either into some lower level, more efficient, virtual-machine code or even into machine code.

Pioneering ideas date back to [Fried58]. However, more recently in [Nord94] the author suggested to directly evolving programs in machine-code form for completely removing the inefficiency in interpreting trees. More recently, a genome compiler has been proposed in [FSMu98], which transforms standard GP trees into machine code before evaluation. The possibilities offered by the Java virtual machine are also currently being explored [KFKB98], [LHMo98].

4.2.4 Directed-Acyclic-Graph Representation

Since the goal is to generate an assembly program, the canonical *S-expression* representation cannot be used. The tree representation was relaxed and the *flow* of the program is represented as a directed acyclic graph (DAG). The semantic of each node in the DAG consists in a pointer to a *macro* inside an *instruction library* and in parameter values. The macro represents a fragment of machine code, usually a single instruction, and parameters represent operand values and registers. It should be remarked that in any assembly language programmers may use several different registers, thus node semantic must include register specification.

A DAG is always translated to a syntactically-correct assembly program. However, it not possible guaranteeing *a-priori* any semantic meaning. An induced program may perform operations on any register, and this exceptional freedom is essential to generate test programs.

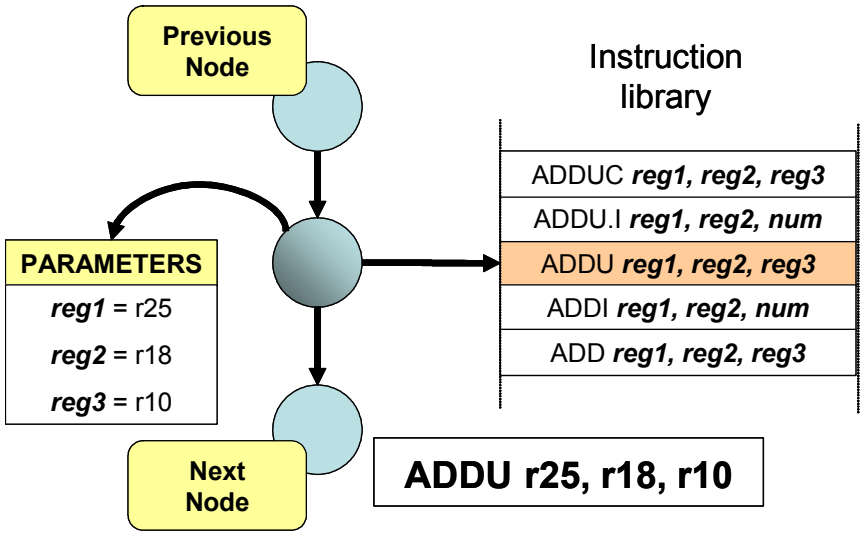

Figure 16: A sequential instruction

Moreover, the library approach has been developed to enable the genetic core and the DAG structure to work with different assembly languages. Different processors not only implement different instruction sets, but also use different formalisms and conventions. Indeed, the method has been successfully tested with three different processors: an i8051, a CISC (complex instruction set computer) micro-controller developed by Intel; a DLX, an academic processor implementing a 5-stage pipeline [HePa] and a SPARC, the well known strongly-pipelined RISC (reduced instruction set computer) processor [18]. Additionally, exploiting abstract macros the program is able to infer data dependencies this ability may be used during assembly-level program generation to avoid inconsistencies.

Each node of the DAG (Figure 17) contains a pointer inside the instruction library and, when needed, its parameters (i.e., immediate values or register specifications). DAGs are built with four kinds of nodes: a *prologue*, an *epilogue*, *sequential instructions*, and *conditional branches*.

Prologue and epilogue nodes are always present and represent required operations, such as initializations. They depend both on the processor and on the operating environment, and they may be empty. The prologue has no parent node, while the epilogue has no children. These nodes may never be removed from the program, nor changed.

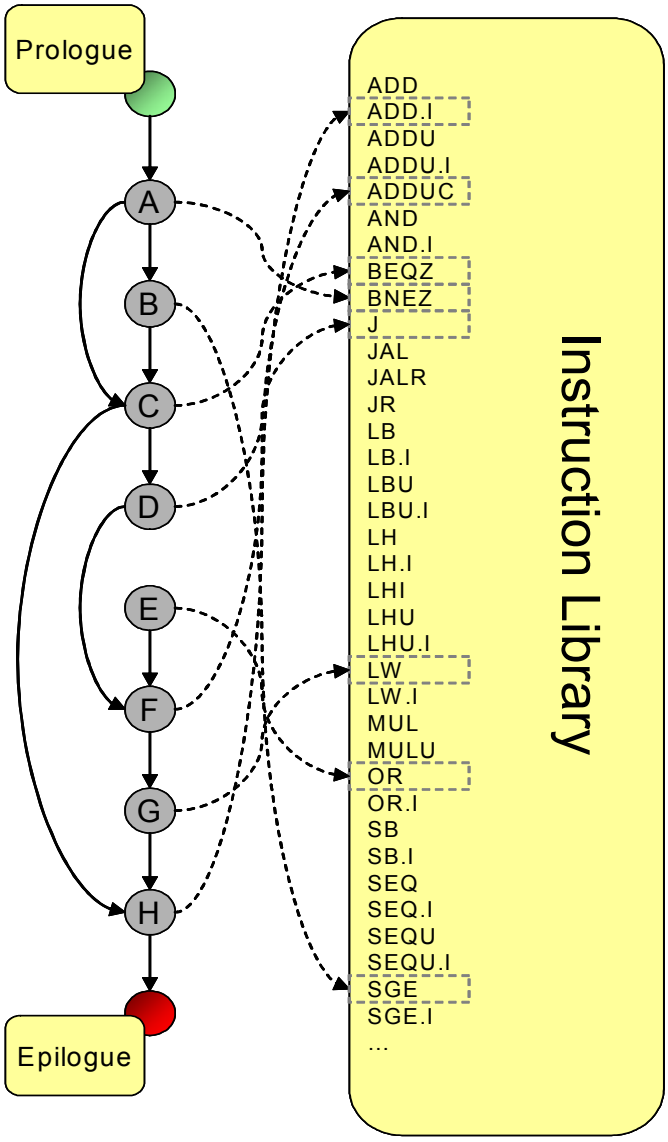

Figure 17: DAG and Instruction library

Sequential-instruction nodes represent common operations, such as arithmetic or logic ones (e.g., node **B**). They have out-degree 1 and the number of parameters changes from instruction to instruction. *Unconditional* branches are considered sequential, since execution flow does not split (e.g., node **D**).

Conditional-branch nodes (e.g., node **A**) are translated to assembly-level conditional-branch instructions. All common assembly languages implement some

jump-if-condition mechanisms. Programmers must use two instructions to check a condition: a test and then a conditional branch.

```
      COMPARE A, B
      JUMP-IF-GREATER g_label
      ;
      ; These operations are executed if A <= B
      ;
g_label:
```

Figure 18: Conditional Branch

Figure 18 reports the assembly pseudo code for a simple *if-then* construct. It's remarkable that, since all these details are masked by compilers, they do not exist in high-level languages.

All conditional branches implemented in the target assembly languages are included in the macro library.

Unconditional branches are treated as sequential operations, since execution flow does not split.

Finally, it must be noted that a single assembly instruction may correspond to more than one entry in the instruction library. For instance, an *ADD* with three registers as operands is distinct from an *ADD* with two registers and one immediate.

Figure 16 show a sequential node that will be translated into an "*ADDU r25, r18, r10*", i.e., store in *r25* the unsigned sum of *r18* and *r10*.

DAG representation does not support backward branches, either conditional or unconditional. This characteristic guarantees program termination, since no endless loop may be implemented. However, the effects of this small reduction in semantic power still need to be evaluated with regards to μP test-program diagnostic effectiveness. As a solution, library will probably include a macro containing a *safe* backward jump, but it will be represented as a sequential node at the DAG level.

4.2.5 Program Induction

Test programs are induced by modifying a DAG topology, by mutating parameters inside a DAG node, or by mating two different DAGs. All modifications are embedded in an evolutionary algorithm implementing a (μ+λ) strategy.

In more details, a population of μ individuals is cultivated, each individual representing a test program. In each step, an offspring of λ new individuals are generated. Parents are selected using tournament selection with tournament size τ (i.e., τ individuals are randomly selected and the best one is picked). Each new individual is generated by applying one or more genetic operators. The cumulative probability of applying at least *n* consecutive operators is equal to p_c^n .

After creating new λ individuals, the best μ programs in the population of (μ+λ) are selected for surviving.

The initial population is generated creating μ empty programs (only prologue and epilogue) and then applying i_m consecutive random mutations to each.

The evolution process iterates until the population reaches a *steady state* condition, i.e., no improvements is recorded for S_g generations.

Three mutation and one crossover operators are implemented and activated with probability p_{add}, p_{del}, p_{mod} and p_{xover} respectively.

- **Add node:** a new node is inserted into the DAG in a random position. The new node can be either a sequential instruction or a conditional branch. In both cases, the instruction referred by the node is randomly chosen. If the inserted node is a branch, either unconditional or conditional, one of the subsequent nodes is randomly chosen as the destination. When an unconditional branch is inserted, some nodes in the DAG may become unreachable (e.g., node **E** in Figure 17).
- **Remove node:** an existing internal node (except prologue or epilogue) is removed from the DAG. If the removed node was the target of one or more branch, parents' edges are updated.
- **Modify node:** all parameters of an existing internal node are randomly changed.

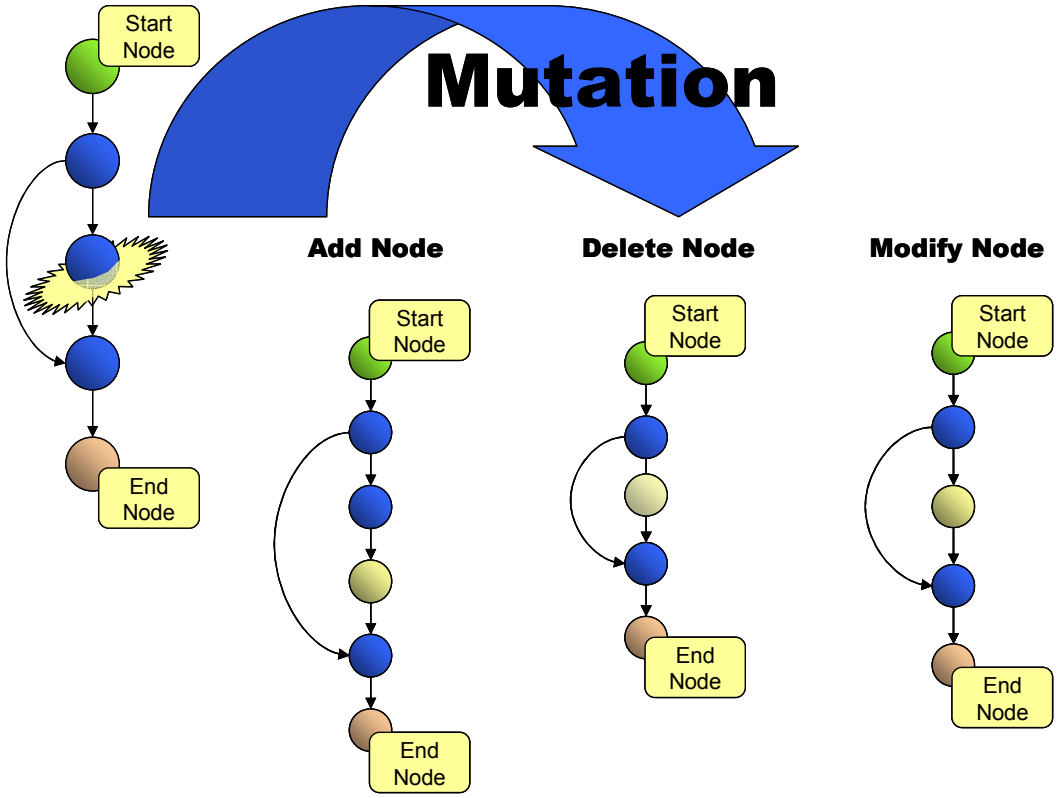

Figure 19: Mutation Operands.

- **Crossover:** two different programs are mated to generate a new one. First, parents are analyzed to detect potential cutting points, i.e., vertices in the DAG that if removed create disjoint sub-graphs (Figure 20). Then a standard 1-point crossover is exploited to generate the offspring.

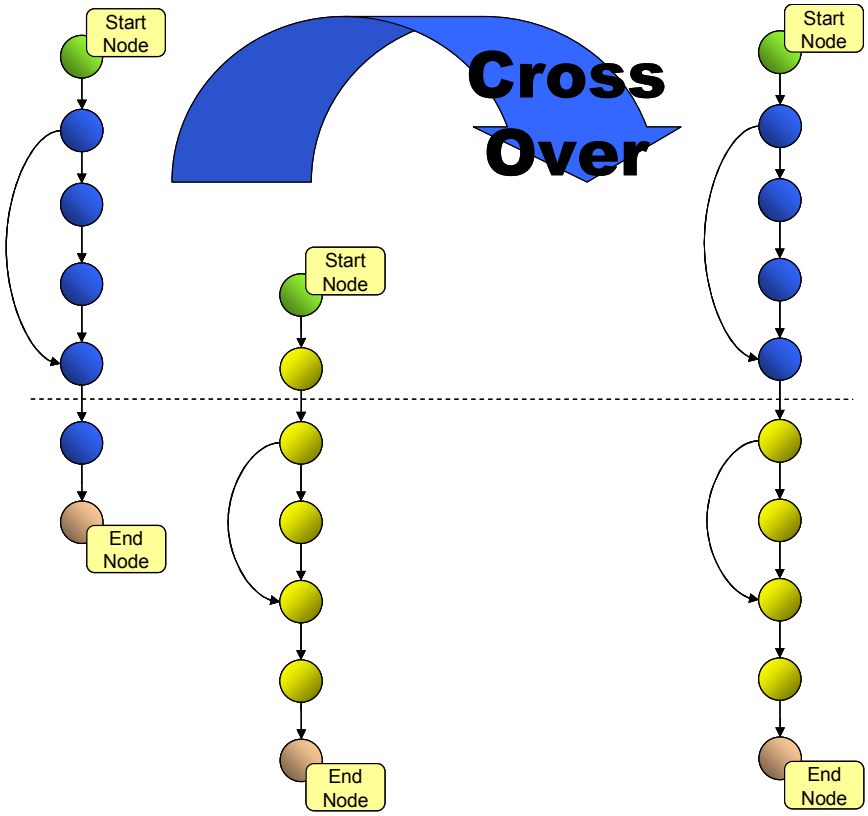

Figure 20: Crossover Operand.

4.2.6 Auto Adaptation

The described approach is able to internally tune both the number of consecutive random mutations and the activation probabilities of all operators. Modifying these parameters, the algorithm is able to shape the search process, significantly improving its performances.

The number of consecutive random mutations is controlled by parameter p_c, which, intuitively, molds the mutation strength in the optimization process. Generally, in the beginning it is better to adopt a high value, allowing offspring to strongly differ from parents. On the other hand, toward the end of the search process, it is preferable to reduce diversity around the local optimum, allowing small mutations only. Initially, the maximum value is adopted ($p_c = 0.9$). Then, the μGP monitors improvements: let I_H be the number of newly created individuals attaining a fitness value higher than their parents over the last H generations. At the end of each generation, the new p_c

value is calculated as $p_c^{new} = \alpha \cdot p_c + (1-\alpha) \cdot \frac{I_H}{H \cdot \lambda}$. Then p_c is saturated to 0.9. The coefficient α introduces inertia to unexpected abrupt changes.

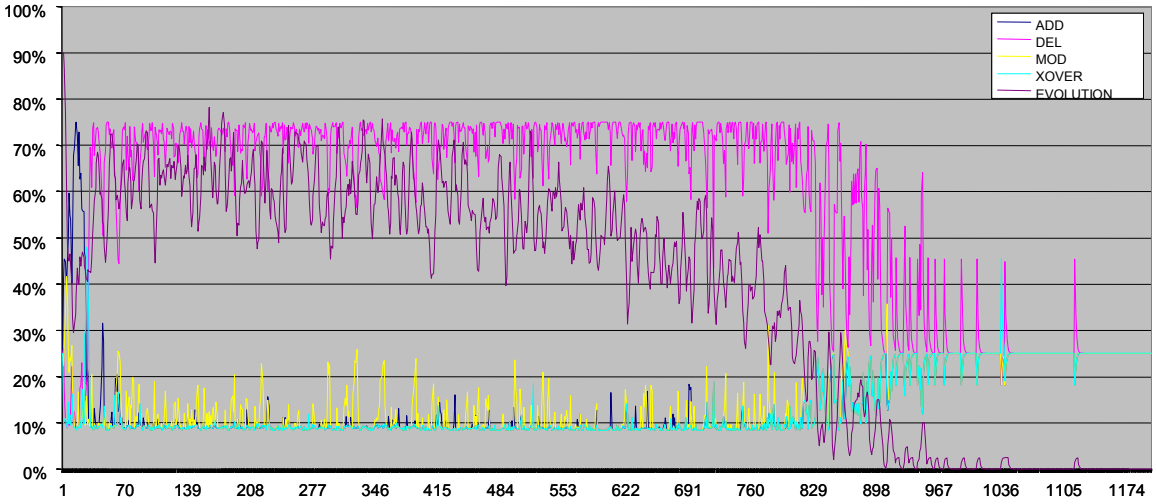

Figure 21: Auto Adaptation.

Regarding activation probabilities, initially they are set to the same value $p_{add} = p_{del} = p_{mod} = p_{xover} = 0.25$. During evolution, probability values are updated similarly to mutation strength: let O_1^{OP} be the number of successful invocation of genetic operator *OP* in the last generation, i.e., the number of invocations of *OP* where the resulting individual attained a fitness value higher than its parents; and let O_1 be the total number of operators invoked in the last generation. At the end of each generation, the new values are calculated as $p_{OP}^{new} = \alpha \cdot p_{OP} + (1-\alpha) \cdot \frac{O_1^{OP}}{O_1}$. Since it is possible that $p_c > 0$, O_1 may be significantly larger than λ. Activation probabilities are forced to avoid values below .01 and over 0.9, then normalized to $p_{add} + p_{del} + p_{mod} + p_{xover} = 1$. If $O_1 = 0$, then all activation probabilities are pushed towards initial values.

Figure 21 show the variation of activation probabilities in a real case.

4.2.7 Experimental Evaluation

Prototypes of the test-program generator and evaluator were implemented in about 3,000 lines of C code. The test-program evaluator exploits *Modelsim* v5.5a by

Model Technology for simulating the design and getting coverage figures, both for RT and gate-levels.

The prototype was tested with one no-pipelined processor (i8051) and two different pipelined processors (DLX/pII and LEON P1754).

4.2.7.1 i8051

Despite its relatively old age, the i8051 is one of the most popular 8-bit micros in use today. Its memory architecture includes 128 bytes of internal data memory that are accessible directly by its instructions. A 32-byte segment of this 128-byte memory block is bit addressable by a subset of the i8051 instructions, namely the bit-instructions.

The i8051 instructions range from 0-operand ones, like *"DIV AB"* (divide accumulator A by B) where all operands are implicit, to 3-operand ones, like "*CJNE Op1, Op2, RelAddr"* (compare *Op1* with *Op2* and jump if they are not equal). The i8051 allows 5 different addressing types: *immediate*, *direct*, *indirect*, *external direct* and *code indirect*. As in many CISC, registers are not orthogonal to the instructions and addressing modes.

The instruction library for the i8051 consists in 81 entries: prologue, epilogue, 66 sequential operations and 13 conditional branches. Listing instructions with their syntax is a trivial task. On the contrary in [CSSV01] preparing the 213 macros required two working days of an experienced engineer.

The methodology was tested on a gate-level implementation consisting in about 12K gates connected to a program memory of 4 Kbytes and a data memory of 2 Kbytes. The complete fault list consists of 28,792 permanent single-bit stack-at faults. A response analyzer was assumed connected to microprocessor output ports and the signature available to the ATE. The test program generator inserts the consequent observability instructions each time a DAG is mapped to an assembly program.

Evaluation has been performed on a Sun Enterprise 250 running at 400 MHz and equipped with 2 Gbytes of RAM. The full generation of the test program required few days, a time comparable with [CSSV01].

Table 13 shows the parameters of the test program generator. They are all standard values and do not require special care. Mutation strength p_c and activation probabilities (p_{add}, p_{del}, p_{mod} and p_{xover}), on the other hand, require careful tuning and are automatically chosen by the algorithm.

PAR	MEANINGS	VALUE
μ	*Population size*	5
λ	*Offspring size*	10
τ	*Tournament size*	2
i_m	*Initial mutations*	100
H	*History for auto-adaption*	4
α	*Auto-Adaption inertia*	0.4
S_g	*Steady state*	500

Table 13: i8051 Test Program Generator Parameters

In order to assess the effectiveness of the approach, the induced test program is compared with a set of selected test programs. Table 14 reports the attained fault coverage.

Fibonacci and *int2bin* are two cross-compiled algorithms. The former calculates the Fibonacci series, while the latter converts an integer to a binary representation. Plausibly, the attained fault coverage is quite low: both are only able to detect 36.04% of the faults.

Approach	FC [%]
Fibonacci	36.04
int2bin	36.04
TestAll	36.35
Random	62.93
Random Macro	80.19
ATPGS	85.19
MicroGP	90.77

Table 14: i8051 Experimental Results

TestAll is an exhaustive functional test program devised by the microprocessor designer, it is relatively long and includes several loops. It tests all possible

instructions; however, since it disregards observability, the fault coverage attained is only slightly superior to former approaches.

Random is the best result attained simulating randomly-generated test programs without evolutionary mechanisms (i.e., selection, mating, survival of the fittest and auto adaptation). For a fair comparison, the same number of programs evaluated by the MicroGP during the generation phase was simulated.

Random Macro corresponds to the results achieved by randomly selecting a sequence of macros created according to [CSSV01]. Results are considerably better than for purely random approach since macros were carefully devised by experts and include sharp mechanisms to make the results observable.

ATPGS reports the result of [CSSV01] where macros and a limited number of evolutionary techniques are exploited. In the approach, a genetic algorithm is given the goal to optimize parameters of heuristically selected macros. Program structures are not evolved, but determined by internal macro code.

MicroGP outperforms all former approaches, reaching a fault coverage of 90.77%. The compilation of the instruction library is a trivial task compared to [CSSV01] and the improvements stems from the enhanced evolutionary mechanisms and the sharper fitness.

A deeper analysis of the fault list enabled identifying a set of combinationally untestable faults in the control unit. Pruning these faults, the fault coverage attained by the MicroGP reaches 94.59%.

4.2.7.2 DLX/pII

Experiments on DLX/pII were targeted at devising test programs for maximizing RT-level statement coverage and gate-level toggle activity.

In more details, the DLX/pII is a 5-stage pipelined version of the DLX microprocessor [PaHe96]. It implements 52 instructions: 18 arithmetic or logic ones, 12 tests, 6 branches, 4 specials, and 12 load/store. DLX supports 3 addressing modes: *register*, *immediate*, *displacement* (i.e., offset). Two additional addressing modes (*register deferred* and *absolute*) may be considered special form of displacement addressing. The exploited architecture is described at the RT-level with 979 VHDL

statements, while the synthesized core is composed of about 38K gates and 650 flip-flops.

The instruction library for the DLX consists in 91 entries: prologue, epilogue, 82 sequential operations and 7 conditional branches. Listing instructions and their syntax was a trivial task. The prologue contains a routine for initializing RAM memory.

Program	INST	CLK	SC [%]	TA[%]
arith_s	47	64	64.56	20.05
carry_su	23	63	65.17	18.22
except	78	108	75.18	17.51
fak	25	72	65.37	16.22
intrpt1	17	217	74.97	12.71
jump1	22	77	66.19	18.58
loadstore_s	129	174	69.25	20.98
loadstore_su	120	174	69.25	19.33
mul_su	15	93	74.67	15.21
set_s	107	144	64.45	20.50
set_su	205	144	64.45	21.04
div32	40	77	64.96	17.64
mul32	24	85	63.94	18.51
System01	199	267	74.36	17.32
all_instr	113	156	79.78	23.51
Cumulative	1,164	1,915	80.59	32.59
μGP_{SC}	812	1,084	94.59	24.38
μGP_{TA}	16,314	19,692	76.00	87.51

Table 15: DLX Summary

Table 15 compares 15 programs in terms of: number of instructions (INST), clock cycles required for execution (CLK), attained RT-level statement coverage (SC) and attained gate-level toggle activity (TA). Programs include functional test programs provided by microprocessor implementers (*arith_s, carry_su, except, fak, intrpt1, jump1, loadstore_s, loadstore_su, mul_su, set_s, set_su*), application (*mul32* and *div32*), system software (*system01*) and an exhaustive functional test that checks all possible instructions (*all_instr*). Row (Cumulative) shows the figures attained cumulatively by all these 15 programs.

Row (μGP_{SC}) reports the result attained by an induced test program which maximizes the RT-level statement coverage. On the other hand, row (μGP_{TA}) shows the results attained by a test program evolved adopting the gate-level toggle activity for evaluating fitness. Devising a test program requires the simulation of about 10,000 programs, corresponding to about two days on a Sun Enterprise 250 with two UltraSPARC-II CPUs at 400MHz, and 2GB of RAM.

For the sake of comparison, 10,000 random programs of about 20K instructions were generated and evaluated. All programs include the prologue and epilogue exploited by μGP. Random test program performance is reported in Table 16 in terms of RT-level statement coverage (SC) and attained gate-level toggle activity (TA). Row (BEST) reports the best result reached by a program, while row (SUM) shows the cumulative figures attained by simulating *all* random programs. Indeed, also simulating 10,000 random programs requires about 2 days on the same workstation.

	SC [%]	TA [%]
BEST	77.12	18.46
SUM	78.87	19.59

Table 16: DLX Random Approach Summary

Firstly, it can be seen that devising a test case able to reach high statement coverage on a pipelined processor is a challenging task, even on a relatively small processor like DLX. Application code is seldom effective to fully validate a design. In fact *mul32*, a 32-bit multiplication performed through shifts and sums, attains the lowest statement coverage. Also specific test benches, like *set_s*, are not able to attain globally good results, despite a long execution time (e.g., *except*).

The induced test program outperforms all other tests, and the versatility of the approach can be seen comparing the last two lines: the use of different fitness functions leads to different results. Maximizing the RT-level statement coverage is different from maximizing the gate-level toggle activity. Interestingly, the latter does not imply the former.

Program	Statement Coverage [%]				
	IF	DEC	EXE	MEM	WB
arith_s	75.15	56.98	100.00	31.69	100.00
carry_su	75.15	58.31	100.00	31.69	100.00
except	84.24	68.51	100.00	49.30	100.00
fak	75.15	58.76	100.00	31.69	100.00
intrpt1	81.21	69.18	100.00	52.11	100.00
jump1	75.15	60.75	100.00	31.69	100.00
loadstore_s	81.21	58.31	100.00	52.11	100.00
loadstore_su	81.21	58.31	100.00	52.11	100.00
mul_su	84.24	67.41	100.00	49.30	100.00
set_s	75.15	56.76	100.00	31.69	100.00
set_su	75.15	56.76	100.00	31.69	100.00
div32	75.15	57.87	100.00	31.69	100.00
mul32	75.15	55.65	100.00	31.69	100.00
system01	84.24	66.74	100.00	49.30	100.00
all_instr	84.24	76.50	100.00	55.63	100.00
Cumulative	87.27	91.57	100.00	59.15	100.00
μGP_{SC}	88.48	96.01	100.00	90.85	100.00
μGP_{TA}	84.24	70.29	100.00	49.30	100.00

Table 17: DLX Statement Coverage Breakdown

Table 17 further details the comparison, showing statement coverage figures for the 5 stages of the pipeline: *instruction fetch* (IF), *decode* (DEC), *execution* (EXE), *memory access* (MEM), and *write-back* (WB). Remarkably, the induced test program outperform all other programs in all stages. The *all_instr* program, carefully designed to exhaustively test *all* possible instructions, is unable to thoroughly verify pipeline stages and attains a statement coverage below 80%. The statement coverage attained by the test program generated by our automatic method is nearly 15% higher. Obviously, further increase in the attained figure may be prevented by the existence of unreachable piece of code in the model.

Considering the gate-level toggle activity, it can be maintained that reaching high figures is an even more challenging task. Designers can hardly foresee the efficacy of a functional test program with regards to gate-level toggle activity. Furthermore, the effect of interactions between simultaneously executed instructions is even more manifest. Executing all instructions, as in the *all_instr*, is far not enough to verify all possible functionalities.

Program	Toggle Activity [%]				
	IF	DEC	EXE	MEM	WB
arith_s	38.35	35.25	9.70	38.86	80.53
carry_su	31.16	32.77	8.25	36.39	82.74
except	35.05	26.77	9.50	37.44	54.87
fak	29.43	26.77	8.88	29.38	36.28
intrpt1	27.91	19.07	7.04	24.29	31.86
jump1	48.95	29.20	10.02	32.95	49.56
loadstore_s	42.03	31.28	10.83	56.60	91.59
loadstore_su	41.19	29.65	10.08	45.19	61.50
mul_su	30.33	24.31	8.48	25.85	34.07
set_s	31.58	35.10	10.95	36.66	82.74
set_su	31.90	36.89	11.02	36.66	82.74
div32	34.99	27.37	9.93	35.29	69.47
mul32	35.26	29.74	9.97	39.37	69.47
system01	34.73	26.44	9.48	36.48	52.65
all_instr	41.61	41.06	10.90	53.71	85.84
Cumulative	55.14	64.01	13.42	65.49	91.59
μGP_{SC}	37.93	46.74	11.39	43.26	82.74
μGP_{TA}	68.26	95.57	86.24	80.61	86.28

Table 18: DLX Toggle Activity Breakdown

Table 18 details the results against pipeline stages. It should be noted that none of the proposed programs is able to toggle more than 50% of the decode stage (DEC), while the induced test program surpasses 95%.

4.2.7.3 LEON P1754

Experiments on LEON target at devising test programs for maximizing RT-level statement coverage, only.

LEON P1754 is the commercial name of a synthesizable VHDL model of the 32-bit SPARC-V8 microprocessor [SPARC]. It was initially developed by the European Space Agency (ESA). The LEON P1754 implements 90 instructions: 20 arithmetic or logic ones, 6 branch, 18 special, 25 load/store and 21 floating point. Only two addressing modes are supported: addresses can be specified either as “register plus register” or “register plus immediate”. The adopted LEON contains a 5-stage pipeline, an internal floating-point unit, and two separate, direct-mapped caches of 2KBytes each for instructions and data. The RT-level description of the microprocessor is about 3K statements long.

Program	INST	CLK	SC[%]
TB	4,964	102,888	73.56
fib100	28	30,072	67.86
random	571	1,264	65.90
μGP	47	531	74.28

Table 19: LEON Statement Coverage summary

The instruction library for the LEON P1754 consists in 230 entries: prologue, epilogue, 118 sequential operations and 110 conditional branches. As above, listing instructions and their syntax was a trivial task.

Unit		Statement Coverage [%]			
Name	STM	FIB	RND	TB	μGP
acache	117	93.16	92.31	94.02	94.87
ahbarb	92	91.30	91.30	91.30	91.30
apbmst	38	100.00	100.00	100.00	100.00
cachemem	30	100.00	100.00	100.00	100.00
dcache	265	73.21	65.28	83.40	84.15
icache	135	89.63	89.63	98.52	98.52
ioport	52	90.38	73.08	98.08	90.38
irqctrl	49	89.80	89.80	95.92	89.80
iu	1,400	59.86	59.57	64.21	67.00
lconf	24	83.33	83.33	83.33	83.33
mcore	31	90.32	90.32	90.32	90.32
mctrl	368	59.51	55.43	63.04	69.84
uart	140	59.29	56.43	85.00	56.43

Table 20: LEON Statement Coverage Breakdown

Table 19 reports 4 programs in terms of: number of instructions (INST), clock cycles required for execution (CLK), and attained RT-level statement coverage (SC). Considered programs include a functional test bench provided by LEON designers (*TB*), a routine for calculating the first 100 numbers in the Fibonacci series (*fib100*), and a test program induced with the proposed approach (*μGP*). For the sake of comparison, 5,000 random programs of 1K instructions were generated and evaluated; the result attained by the best one is reported in row (*random*). For devising the test program, the μGP requires the simulation of about 5,000 programs, corresponding to about two days on a Sun Enterprise 250 with an UltraSPARC-II CPU at 400MHz, and 2GB of RAM.

The first striking fact is that the induced test program is able to attain a statement coverage slightly superior to the one achieved by the test bench using 10% of the instructions and 0.5% of the execution time.

Table 20 details statement coverage figures against the main modules of LEON: interface between I/D cache controllers and AMBA Advanced High-speed bus (*acache*); AMBA arbiter and decoder (*ahbarb*); AMBA AHB/APB bridge (*apbmst*); ram cells for both instruction and data caches (*cachemem*); data cache controller (*dcache*); instruction cache controller (*icache*); parallel I/O port (*ioport*); interrupt controller (*irqctrl*); integer unit (*iu*); configuration register (*lconf*); standard peripherals and LEON core (*mcore*); external memory controller (*mctrl*); asynchronous UART (*uart*).

The induced program shows a lower efficacy on the *irqctrl* block than *TB*. This can be easily explained, since interrupts can not be induced by the actual version of the test-program generator. Indeed, induced test-program efficacy on *irqctrl* is equal to *fib100* and *random*. On the *uart* the result attained by the induced program is superior to both *fib100* and *random*, but inferior to the one attained by the test bench. Possibly, the generator was not given enough time to devise more efficient programs. On the other hand, the induced test program attains the best results testing the cache interface, the integer unit, and the external memory controller, showing its capability to handle complex interactions inside pipeline stages.

5 Conclusions

Due to the wide adoption of logic synthesis tools, RT-level ATPG techniques are increasingly important in order to shift test-related activities towards the description level adopted by designers. A crucial point for developing effective high-level ATPGs lies in the identification of a suitable fault model, which should guarantee a good correlation with gate-level fault coverage figures while allowing the implementation of an ATPG algorithm.

In Chapter 2 the RT-level Single-bit Stuck-at fault model is described. This fault model, thanks to a careful identification of redundancies removed by synthesis, is shown to be highly correlated with Gate-level fault coverage. This allows designers to predict circuit testability before synthesis.

In the same Chapter is described an approach to fault simulation of RT-level description based on exploiting the existing debug mechanisms of commercial VHDL simulators. With a relatively moderate effort, an effective fault simulator can be built by properly programming a VHDL simulator. The implemented fault simulation system is able to simulate a refined version of the widely used observability-enhanced statement coverage metric, where observability is explicitly taken into account in an exact manner.

Experimental results prove the feasibility of the approach, and show that access to the source code of a VHDL simulator or modifications of the VHDL code are not necessary in order to compute faulty responses from a digital circuit. The efficiency of VHDL simulation cores and the versatility of their user interfaces open up the possibility of greatly optimizing the efficiency of this approach.

Another crucial point for developing effective high-level test signal generator is the availability of a suitable algorithm for test generation.

In Chapter 3 the RT-level Single-bit Stuck-at fault model is exploited to describe a methodology for generating high quality test signals: ARPIA. The approach exploits an evolutionary algorithm to drive the search of effective patterns within the gigantic space of all possible signal sequences. ARPIA operates on register-transfer level VHDL descriptions and generates effective test patterns.

Being an evolutionary algorithm, ARPIA evolves a population seeking fitter individuals. But, since individuals are test sequences for a digital circuit, the fitness measures the sequence ability to detect faults in the design. And it is computed by simulation. Given a fault model, the *fault coverage* is defined as the percentage of faults that the test sequence is able to detect. Thus, the goal of ARPIA can be rephrased as "generate a sequence of signals that attains maximum fault coverage."

Experimental results on the ITC99 RT-level benchmarks show that the achieved results are comparable or better than those obtained by high-level similar approaches or even by gate-level ones.

The issue of System On a Chip (SOC) testing is one of the most crucial in their design and production process. A popular solution for SOCs including microprocessor cores is based on letting them execute a test program, thus implementing a very attracting BIST solution. Chapter 4 describes two methodologies for generating test programs for microprocessors and microcontrollers: a semi-automatic and an automatic one.

The semi-automatic methodology (Section 4.1) requires the availability of a small library of macros, whose development should be performed by hand, based on the mere knowledge of the instruction set.

The main novelty of this approach lies in the fact that it only relies on the RT-level description of the device, and does not exploit any knowledge about lower-level implementation details. An optimization algorithm is outlined for selecting the minimal subset of macros, and their parameters. The algorithm entirely works on the RT-level description, exploiting a suitable RT-level fault model.

Experimental results gathered on the Intel 8051 microcontroller using a prototypical implementation of the method show that the generated test program attains higher fault coverage figures (in terms of gate-level stuck-at faults) than the

test program generated starting from the gate-level description, with a comparable computational effort, thus demonstrating the practical viability of the approach.

Section 4.2 describes the automatic methodology, an efficient and versatile approach to test-program generation based on an evolutionary algorithm. This methodology is able to automatically induce an assembly test program for microprocessor tackling complex pipelined designs. The methodology exploits a loose coupling between a generator and an evaluator, and can be used to induce test programs with different goals, i.e., maximizing diverse verification metric.

First, the method includes the ability to explicitly specify registers either as operands or targets. Furthermore, it relaxes the usual tree-based representation, resorting to DAG. Finally, it couples a standard GP approach with a database containing the assembly-level semantic associated to DAG nodes.

Exploiting DAG and library the proposed approach is extremely efficient and versatile. Candidate solutions are translated into source-code programs that can be assembled and linked using standard compilers. Such executable programs allow a fast evaluation: millions of runs may be performed in just a second.

Moreover, the approach is versatile. The instructions library allows changing the target μP and environment easily. The approach was verified on three processors with different instruction sets, different formalisms and different conventions.

This automatic methodology can be seen as a general enhancement of standard GP; however it was specifically devised for inducing test-program for μP cores, a critical area in modern industry.

Prototypes of both generator and evaluator were built and exploited to generate test programs against two pipelined microprocessors: a DLX/pII (a 5-stage pipelined implementation of the DLX microprocessor) and a LEON P1754 (a 5-stage pipelined, synthesizable, 32-bit SPARC-V8 processor). Test programs were devised trying to maximize two different metrics: RT-level statement coverage and gate-level toggle activity.

Induced test programs outperformed other approaches and surpass the (supposedly) exhaustive test benches provided by designers. Thus, in acceptable

computation effort, engineers could get high-quality test programs exploitable in a simulation-based verification process.

6 References

[AABH99] J. Shen, J. Abraham, D. Baker, T. Hurson, M. Kinkade, "Functional verification of the Equator MAP1000 microprocessor", *36th Design Automation Conference*, 1999, pp. 169 -174

[ABFr90] M. Abramovici, M. A. Breuer, A. D. Friedman, *Digital systems testing and testable design*, Computer Science Press, 1990

[BaPa99] K. Batcher, C. Papachristou, "Instruction Randomization Self Test For Processor Cores", *IEEE VLSI Test Symposium*, 1999, pp. 34-40

[Beiz90] B. Beizer, Software Testing Techniques (2nd ed.), Van Nostrand Rheinold, New York, 1990

[BHSc91] T. Bäck, F. Hoffmeister, and H.-P. Schwefel, "A survey of evolution strategies", *Proceedings of the Fourth International Conference on Genetic Algorithms*, 1991, pp. 2-9

[BiMa95] U. Bieker and P. Marwedel, "Retargetable self-test program generation using constraint logic programming," *32nd Design Automation Conference*, 1995, pp. 605 – 611

[CCSS00] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "An RT-level Fault Model with High Gate Level Correlation," *IEEE International High Level Design Validation and Test Workshop*, November 8-10, 2000

[CCSS00a] F. Corno, G. Cumani, M. Sonza Reorda, Giovanni Squillero, "RT-level Fault Simulation Techniques based on Simulation Command Scripts," *DCIS 2000: XV Conference on Design of Circuits and Integrated Systems*, November 21-24, 2000

[ChDe00] L. Chen, S. Dey, "DEFUSE: A Deterministic Functional Self-Test Methodology for Processors", IEEE VLSI Test Symposium, 2000, pp. 255-262

[CPRS96] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, "GATTO: a Genetic Algorithm for Automatic Test Pattern Generation for Large Synchronous Sequential Circuits", *IEEE Transactions on Computer-Aided Design*, August 1996, Vol. 15, No. 8, pp. 943-951

[CSSq00] F. Corno, M. Sonza Reorda, G. Squillero, "Exploiting ITC'99 benchmarks for developing an RT-level ATPG tool", *IEEE Design & Test, Special issue on Benchmarking for Design and Test,* June 2000, pp. 44-53

[CSSq00a] F. Corno, M. Sonza Reorda, G. Squillero, "High-Level Observability for Effective High-Level ATPG," *VTS2000: 18th IEEE VLSI Test Symposium*, May 2000, pp. 411-416

[CSSq01] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante, "On the Test of Microprocessor IP Cores", *DATE*, IEEE Design, Automation & Test in Europe Conference, 2001, pp. 209-213

[CSSV01] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante, "On the Test of Microprocessor IP Cores", *IEEE Design, Automation & Test in Europe*, 2001, pp. 209-213

[DGke96] S. Devadas, A. Ghosh, K. Keutzer, "An Observability-Based Code Coverage Metric for Functional Simulation," *Proceedings IEEE/ACM International Conference on Computer Aided Design*, 1996

[DGKe96] S. Devadas, A. Ghosh, K. Keutzer, "An Observability-Based Code Coverage Metric for Functional Simulation," Proceedings IEEE/ACM International Conference on Computer Aided Design, 1996

[FADe99] F. Fallah, P. Ashar, S. Devadas, "Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage," Proceedings 35th Design Automation Conference, 1999, pp. 666-671

[FDKe98] F. Fallah, S. Devadas, K. Keutzer, "OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification," *DAC98: 34th Design Automation Conference*, 1998

[FFFFS01] G. Ferrara, F. Ferrandi, A. Fin, F. Fummi, D.Sciuto, "Functional Test Generation for Behaviorally Sequential Models", *Proceedings IEEE Design Automation and Test in Europe Conference (DATE)*, Muenchen, Germany, 13-16 Marc 2001, pp.403-410.

[FFGS99] F. Ferrandi, F. Fummi, L. Gerli, D. Sciuto, "Symbolic Functional Vector Generation for VHDL Specifications," *DAC99: 35th Design Automation Conference*, 1999, pp. 442-446

[FFSc98] F. Ferrandi, F. Fummi, D. Sciuto, "Implicit Test Generation for Behavioral VHDL Models," Proceedings IEEE International. Test Conference, 1998

[FiFu00] A. Fin, F. Fummi, "A VHDL Error Simulator for Functional Test Generation," *IEEE European Design, Automation and Test Conference*, 2000, pp. 390-395

[Fried58] R. M. Friedberg, "A Learning Machine: Part {I)", IBM Journal of Research and Development, 1958, vol. 2, n. 1, pp 2-13

[FSMu98] A. Fukunaga, A. Stechert, D. Mutz, "A genome compiler for high performance genetic programming", Genetic Programming 1998: Proceedings of the 3rd Annual Conference, 1998, pp. 86-94

[GSHA00] A. Giani, S. Sheng, S. Hsiao, I. Agrawal, "Compaction-Based Test Generation Using State and Fault Information", in *Proceedings Asian Test Symposium*, pp. 159-164, 2000

[Hand94] S. Handley, "On the use of a directed acyclic graph to represent a population of computer programs", *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, 1994, pp 154-159

[HeDu01] J. R. Heath, S. Durbha, "Methodology for synthesis, testing, and verification of pipelined architecture processors from behavioral-level-only HDL code and a case study example", *Proceedings IEEE SoutheastCon 2001*, 2001, pp. 143-149

[HePa] J.L. Hennessy, D.A. Patterson, "DLX architecture", in *Computer Architecture, a Quantitative Approach*, Morgan Kaufmann Publishers.

[KFKB98] S. Klahold, S. Frank, R. E. Keller, W. Banzhaf, "Exploring the possibilites and restrictions of genetic programming in Java bytecode", *Late Breaking Papers at the Genetic Programming 1998 Conference*, 1998

[Koza98] J. R. Koza, "Genetic programming", *Encyclopedia of Computer Science and Technology*, vol. 39, Marcel-Dekker, 1998, pp. 29-43

[KPGZ02] N. Kranitis, A. Paschalis, D. Gizopoulos, Y. Zorian, "Effective software self-test methodology for processor cores", *IEEE Design, Automation & Test in Europe*, 2002, pp. 592-597

[LeSi91] D. C. Lee, D. P. Siewiorek, "Functional test generation for pipelined computer implementations", *21st International Symposium on Fault-Tolerant Computing*, 1991, pp. 60 67

[LHMo98] E. Lukschandl, M. Holmlund, E. Moden, "Automatic evolution of Java bytecode: First experience with the Java virtual machine," *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming, 1998*, pp 14-16

[NCPa92] T.M. Niermann, W.-T. Cheng, J.H. Patel, "PROOFS: A Fast, Memory-Efficient Sequential Circuit Fault Simulator", *IEEE Trans. on CAD/ICAS*, Vol. 11, No. 2, February 1992, pp. 198-207

[Nord94] P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," *Advances in Genetic Programming*, 1994, pp. 311-331

[PaHe96] D. A. Patterson and J. L. Hennessy, *Computer Architecture - A Quantitative Approach, (second edition)*, Morgan Kaufmann, 1996

[PITC99] High Time for High-Level Test Generation*, Panel at the IEEE International Test Conference*, 1999, pp. 1112-1119

[PJAV02] L.M. Patnaik, H.S. Jamadagni, V.K. Agrawal, B.K.S.V.L. Varaprasad, "The state of VLSI testing", IEEE Potentials , Volume: 21 Issue: 3 , 2002, pp. 12 -16

[PMNo99] C.A. Papachristou, F. Martin, M. Nourani, "Microprocessor Based Testing for Core-Based System on Chip",*ACM/IEEE Design Automation Conf*erence, 1999, pp. 586-591

[Poli97] R. Poli, "Evolution of graph-like programs with parallel distributed genetic programming", *Genetic Algorithms: Proceedings of the 7th International Conference*, 1997, pp 346-353

[RiUc96] T. Riesgo, J. Uceda, "A Fault Model for VHDL Descriptions at the Register Transfer Level," Proceedings of EURO-DAC/EURO-VHDL, 1996

[RLJh98] S. Ravi, G. Lakshminarayana, and N. K. Jha, "TAO: Regular expression based high-level testability analysis and optimization", in *Proceedings International Test Conference*, pp. 331–340, 1998

[SATa96] A. E. Salama, A.K. Ali, E. A. Talkhan, "Functional testing of pipelined processors", *IEE Proceedings on Computers and Digital Techniques*, vol. 143, issue 5, September 1996, pp. 318-324

[ScKu98] H.-P. Schwefel, F. Kursawe, "On Natural Life's Tricks to Survive and Evolve", *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*, 1998, pp. 1-8

[SGTT00] M. B. Santos, F. M. Gonçalves, I.C. Texeira, J. P. Texeira, "RTL-Based Functional Test Generation for High Defects Coverage in Digital SOCs", *IEEE European Test Workshop*, 2000, pp. 99-104

[ShAb98] J. Shen and J.A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation", *International Test Conference*, 1998, pp. 990-999

[ShSu88] L. Shen, S. Y. H. Su, "A functional testing method for microprocessors", *IEEE Transactions on Computers*, vol. 37, issue 10, October 1988, pp. 1288-1293

[SPARC] SPARC International, *The SPARC Architecture Manual*

[TAZa99] P. A. Thaker, V. D. Agrawal, M. E. Zaghloul, "Validation Vector Grade (VVG): A New Coverage Metric fo Validation and Test," Proceedings 15th IEEE VLSI Test Symposium, 1997, pp. 182-188

[ThAb80] S. Thatte, J. Abraham, "Test Generation for Microprocessors", *IEEE Transactions on Computers*, Vol. C-29, June 1980, pp. 429-441

[UBSh99] N. Utamaphethai, R.D. Blanton and J.P. Shen, "Superscalar Processor Validation at the Microarchitecture Level", *12th IEEE International Conference on VLSI Design*, 1999, pp. 300-305