Fully Automatic Test Program Generation for Microprocessor Cores

Abstract

Microprocessor cores are a major challenge in the test arena: not only is their complexity always increasing, but also their specific characteristics intensify all difficulties. A microprocessor embedded inside a SOC is even harder to test since its input might be harder to control and its behavior may be harder to observe. Functional testing, an effective solution, consists in forcing the microprocessor to execute a suitable test program. This paper presents a new approach to automatic test program generation, which overcomes the main limitations of previous methodologies and provides significantly better results. Human intervention is limited to the enumeration of all assembly instructions with their possible operands. Also internal parameters of the optimizer are auto-adapted. Experimental results show the effectiveness of the approach.

1 Introduction

Developments in semiconductor technology have made possible to design an entire system onto a single chip, the so-called *System-on-a-Chip* (SOC). A connected practice is the usage of predefined logic blocks known as *cores* or, sometimes, *macros*. 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.

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 (DFT) 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 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 paper presents a new approach to test program generation overcoming the main limitations of previous approaches and providing significantly better results. The method exploits evolutionary techniques to automatically induce a very effective test program. Assembly programs are internally represented as directed acyclic graphs and evolved to maximize the attained fault coverage. Concurrently, internal parameters are adapted to their optimal values. Human intervention is limited to the enumeration of all available instructions and their possible operands.

Experiments gathered exploiting a prototypical tool on the i8051 microprocessor show that, despite the much lower human intervention, the method is able to overcome the results of [CSSV01] in term of attained fault coverage.

The paper is organized as follows. Section 2 details test-program generation and adopted evolutionary techniques. Section 3 reports the experimental evaluation, while Section 3 draws some conclusions

2 Test-Program Generation

This paper proposes an automatic methodology for generating a test program able to attain high fault coverage figures. Indeed, creating in an automated way programs able to solve problems is a fascinating task and has always been one of the more desired goals of computer science.

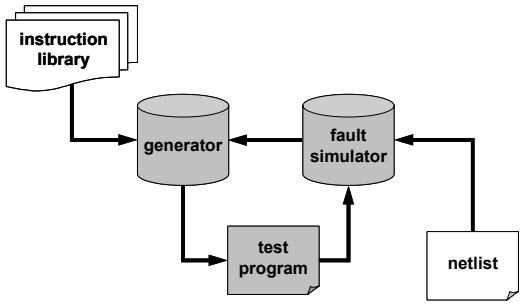

Figure 1: *System Architecture*

The overall architecture of the proposed approach is shown in Figure 1. The *generator* induces test programs exploiting an external *instruction library* that describes the syntax of the microprocessor assembly language. The generator utilizes a fault simulator to evaluate the generated test programs and to gathered relevant information for driving the optimization process.

Next Sections detail the approach.

2.1 Representation

Test program generation exploits MicroGP, an approach for inducing assembly programs able to reach a specific goal. MicroGP was theoretically developed in [CCSS02a]. It utilizes a directed acyclic graph (DAG) for representing the flow of a program, and an instruction library for describing the assembly syntax. The loose coupling between these two elements enables

exploiting the approach with different instruction sets, formalisms and conventions.

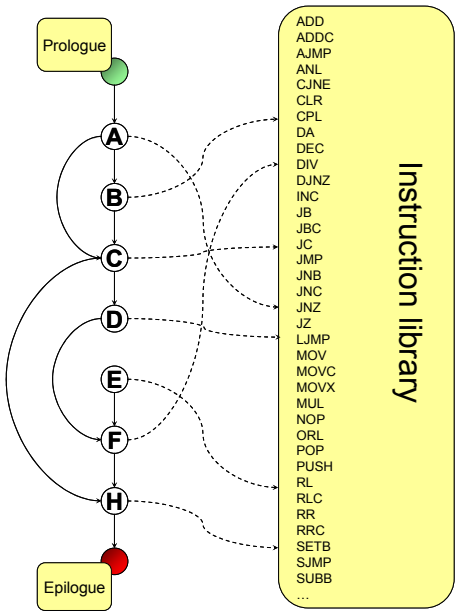

Figure 2: DAG and Instruction library

Each node of the DAG (Figure 2) contains a pointer inside the instruction library and, when needed, its parameters (i.e., immediate values or register specifications). For instance, Figure 3 shows a sequential node that will be translated into an "*ORL A, R1*", i.e., a bit-wise OR between accumulator and register R1. DAGs are built with four kinds of nodes:

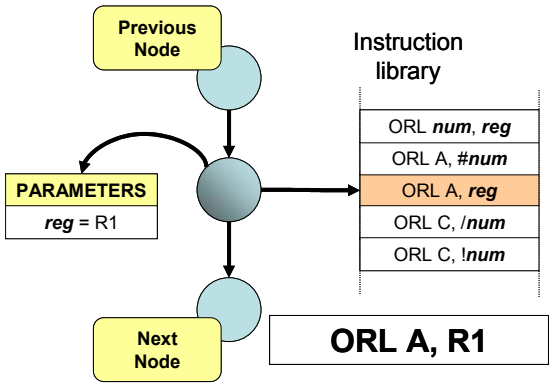

Figure 3: *A sequential instruction*

- **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.
- **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 are translated to assembly-level conditional-branch instructions (e.g., node **A**). All conditional branches implemented in the target assembly languages must be included in the library.

2.2 Program Induction

Test programs are induced by modifying DAG topology and by mutating parameters inside DAG nodes. Both kinds of 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 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 are 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. Remarkably, when an unconditional branch is inserted, some nodes in the DAG may become unreachable (e.g., node **E** in Figure 2).
- **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.
- **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 (e.g., node **C** in Figure 2). Then a standard 1-point crossover is exploited to generate the offspring.

2.3 Program Evaluation

Test program evaluation associates a fitness value to each test program. Biological fitness measures the capacity of an organism to survive and transmit its genotype to reproductive offspring as compared to competing organisms. For the purpose of this paper, the fitness of a test program measures its efficacy (attained fault coverage) and its potential (ability to excite new faults).

Fitness values are 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.

After translating the DAG into a syntactically correct assembly program, the execution of the test case is fault simulated. Three values are gathered: the number of excited faults (N_a); the number of faults that modified at least a memory element (N_m); the number of detected faults (N_d). Let N_{tot} be the total number of faults, the fitness is calculated as: $f = N_a + N_{tot} \cdot N_m + (N_{tot})^2 \cdot N_d$.

Faults are marked detected coherently with the test solution adopted, e.g., a signature into a specified memory position available to the ATE through DMA, a linear feedback shift register (LFSR) monitoring buses, a response analyzer connected to microprocessor outputs, or other architectures.

2.4 Auto Adaptation

The proposed 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 MicroGP 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.

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.

3 Experimental Results

A prototype of the proposed test program generator was implemented in about 3,000 lines of C code. For evaluating test programs an in-house developed fault-parallel event-driven fault simulator is used.

The methodology was tested on an i8051 core. 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 1 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 1: 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 2 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 2: *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 Conclusions

Software BIST methodologies present several advantages over scan-based ones, they do not require high-cost ATE equipment, and they allow an at-speed test of the processor core. Furthermore on a microprocessor core the insertion of test architectures may be unpractical or incompatible with other SOC-level DFT structures.

In this paper we presented a new approach for devising test programs for a microprocessor core. The methodology exploits innovative evolutionary techniques to direct the search process.

The test program generator utilizes a directed acyclic graph for representing the flow of an assembly program, and an instruction library for describing the assembly syntax. The loose coupling between these two elements enables exploiting the approach with different instruction sets, formalisms and conventions.

Moreover, concurrently to test program generation, the evolutionary core adapts its internal parameters, reducing even more the required user effort.

Experiments gathered on the i8051 microprocessor exploiting a prototypical tool show that the proposed approach is able to reach higher fault coverage with fewer limitations than alternative approaches.

In contrast to other methodologies, human intervention is limited to the enumeration of all available instructions and their possible operands. Indeed, the experimented test program generation task required an instruction library of 81 entries, while in [CSSV01], for the same processor, an experienced engineer needs two working days to prepare the set of 213 macros.

Furthermore, the ability to generate assembly level programs for a generic microprocessor may be exploited for different purposes. In [CCSS02b], a similar approach was used against a register-level VHDL description for optimizing code coverage metrics.

Current work is targeted to apply the proposed approach to more complex processors, with enhanced features such as caches and pipelines. The first results are already available on a DLX/pII [PaHe96], an academic microprocessor implementing a 5-stage pipeline, and a SPARC V8 [SPARC].

5 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

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

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

[CCSS02a] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Efficient Machine-Code Test-Program Induction", *Congress on Evolutionary Computation*, 2002, pp. 1486-1491

[CCSS02b] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Evolutionary Test Program Induction for Microprocessor Design Verification", to appear in: *11th Asian Test Symposium*, 2002

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

[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

[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

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

[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

[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

[SPARC] SPARC International, The SPARC Architecture Manual

[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