Evolution of Test Programs Exploiting a FSM Processor Model

Evolution of Test Programs Exploiting a FSM Processor Model

Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda

Dipartimento di Automatica e Informatica
Politecnico di Torino, Italy
{ernesto.sanchez,giovanni.squillero,alberto.tonda}@polito.it

Abstract. Microprocessor testing is becoming a challenging task, due to the increasing complexity of modern architectures. Nowadays, most architectures are tackled with a combination of scan chains and Software-Based Self-Test (SBST) methodologies. Among SBST techniques, evolutionary feedback-based ones prove effective in microprocessor testing: their main disadvantage, however, is the considerable time required to generate suitable test programs.

A novel evolutionary-based approach, able to appreciably reduce the generation time, is presented. The proposed method exploits a high-level representation of the architecture under test and a dynamically built Finite State Machine (FSM) model to assess fault coverage without resorting to time-expensive simulations on low-level models. Experimental results, performed on an OpenRISC processor, show that the resulting test obtains a nearly complete fault coverage against the targeted fault model.

Keywords: SBST microprocessor testing.

1 Introduction

In the last years, the market demand for a higher computational performance in embedded devices has been continuously increasing for a wide range of application areas, from entertainment (smart phones, portable game consoles), to professional equipment (palmtops, digital cameras), to control systems in various fields (automotive, industry, telecommunications). The largest part of today's Systems-on-Chip (SoCs) includes at least one processor core. Companies have been pushing design houses and semiconductor producers to increase microprocessor speed and computational power while reducing costs and power consumption. The performance of processor and microprocessor cores has impressively increased due to technological and architectural aspects. Microprocessor cores are following the same trend of high-end microprocessors and quite complex units may be easily found in modern SoCs.

Technology advancements impose new challenges to microprocessor testing: as device geometries shrink, deep-submicron delay defects are becoming more prominent [5], thereby increasing the need for at-speed tests; as core operating frequency and speed of I/O interfaces rise, more expensive external test equipment is required.

The increasing size and complexity of microprocessor architectures directly reflects in more demanding test generation and application strategies. Modern designs contain intricate architectures that increase test complexity. Indeed, pipelined and superscalar designs demonstrated to be random pattern resistant [2]. The use of hardware-based approaches, such as scan chains and BIST, even though consolidated in industry for integrated digital circuits, has proven to be often inadequate, since these techniques introduce excessive area overhead [1], require extreme power dissipation during the test application [13], and are often ineffective when testing delay-related faults [12].

As a consequence, academy is looking for novel paradigms to respond to the new testing issues: one promising alternative to hardware-based approaches is to exploit the processor to execute carefully crafted test programs. The goal of these test programs is to uncover possible design or production flaws in the processor. This technique, called Software-Based Self-Test (SBST), has been already used in different problems with positive results.

In this paper, we propose a SBST simulation-based framework for the generation of post-production test programs for pipelined processors. The main novelty of the proposed approach is its ability to efficiently generate test programs, exploiting a high level description of the processor under test, while the evolution of the generation is driven by the transition coverage of a FSM created during the evolution process itself.

The rest of the paper is organized as follows: section 2 provides background on SBST and SBST-related evolutionary computation. Section 3 describes the proposed methodology. Section 4 outlines the case study and reports the experimental results, and section 5 drafts some conclusions of our work.

2 Background

2.1 SBST

SBST techniques have many advantages over other testing methodologies, thanks to their features: the testing procedure can be conducted with very limited area overhead, if any; the average power dissipation is comparable with the one observable in mission mode; the possibility of damages due to excessive switching activity, non-negligible in other methods, is virtually eliminated; test programs can be run at the maximum system speed, thus allowing testing of a larger set of defects, including delay-related ones; the approach is applicable even when the structure of a module is not known or cannot be modified.

SBST approaches proposed in literature do not necessarily aim to substitute other established testing approaches (e.g., scan chains or BIST) but rather to supplement them by adding more test quality at a low cost. The objective is to create a test program able to run on the target microprocessor and test its modules, satisfying the target fault coverage requirements. Achieving this test quality requires a proper test program generation phase, which is the main focus of most SBST approaches in recent literature. The quality of a test program is measured by its coverage of the design errors or production defects, its code size, and time required for its execution.

The available approaches for test program generation can be classified according to the processor representation that is employed in the flow. High-level representations

of the processor Instruction Set Architecture (ISA) or state transition graphs are convenient for limiting the complexity of the architecture analysis, and provide direct correlation with specific instruction sequences, but cannot guarantee the detection of structural faults. Lower-level representations, such as RT and gate-level netlists, describe in greater detail the target device and allow to focus on structural fault models, but involve additional computational effort.

A survey on some of the most important techniques developed for test program generation is presented in [9]. Due to modern microprocessors' complex architectures, automatic test program generation is a challenging task: considering different architectural paradigms, from pipelined to multithreaded processors, the search space to be explored is even larger than that of classic processors. Thus, it becomes crucial to devise methods able to automate as much as possible the generation process, reducing the need for skilled (and expensive) human intervention, and guaranteeing an unbiased coverage of corner cases.

Generation techniques can be classified in two main groups: *formal* and *simulation-based*. Formal methodologies exploit mathematical techniques to prove specific properties, such as the absence of deadlocks or the equivalence between two descriptions. Such a proof implicitly considers all possible inputs and all accessible states of the circuit. Differently, simulation-based techniques rely on the simulation of a set of stimuli to unearth misbehaviors in the device under test. A simulation-based approach may therefore be able to demonstrate the presence of a bug, but will never be able to prove its absence: however, the user may assume that the bug does not exist with level of confidence related to the quality of the simulated test set. The generation of a qualifying test set is the key problem with simulation-based techniques. Different methodologies may be used to add contents to such test sets, ranging from deterministic to pseudo-random.

Theoretically, formal techniques are able to guarantee their results, while simulation-based approaches can never reach complete confidence. However, the former require considerable computational power, and therefore may not be able to provide results for a complex design. Moreover, formal methodologies are routinely applied to simplified models, or used with simplified boundaries conditions. Thus, the model used could contain some differences with respect to the original design, introducing a certain amount of uncertainty in the process [8].

Nowadays, simulation-based techniques dominate the test generation arena for microprocessors, with formal methods bounded to very specific components in the earliest stages of the design. In most of the cases, simulation-based techniques exploit *feedback* to iteratively improve a test set in order to maximize a given target measure. Nevertheless, simulation of low-level descriptions could require enormous efforts in terms of computational time, memory occupation and hardware.

The main drawback of feedback-based simulation methods, is the long elaboration time required during the evolution. When dealing with a complete processor core, for example in [10], the generation time increases when low abstraction descriptions are used as part of the evolution: the growth of computation times is mainly due to the inclusion of *fault simulation* in the process. For every possible fault of the design, the original circuit is modified including the considered fault; then, a complete simulation is performed in order to understand whether the fault changes the circuit

outputs. Even though lots of efforts are spent on improving this process [6], several minutes are still required to perform a fault simulation on a processor core with about 20k faults.

2.2 EAs on SBST

Several approaches that face test program generation by exploiting an automated methodology have been presented in recent years: in [11] a tool named VERTIS, able to generate both test and verification programs based on the processor's instruction set architecture only, is proposed. VERTIS generates many different instruction sequences for every possible instruction being tested, thus leading to very large test programs. The test program generated for the GL85 processor following this approach is compared with the patterns generated by two Automatic Test Pattern Generator (ATPG) tools: the test program achieves a 90.20% stuck-at fault coverage, much higher than the fault coverage of the ATPG tools, proving the efficacy of SBST for the first time. The VERTIS tool works with either pseudo-random instruction sequences and random data, or with test instruction sequences and heuristics to assign values to instruction operands specified by the user in order to achieve good results. In more complex processors, devising such heuristics is obviously a non-trivial task.

In [7], an automated functional self-test method, called *Functional Random Instruction Testing at Speed* (FRITS), is presented. FRITS is based on the generation of random instruction sequences with pseudorandom data. The authors determine the basic requirements for the application of a cache-resident approach: the processor must incorporate a cache load mechanism for the test program downloading and the loaded test program must not produce either cache misses or bus cycles. The authors report some results on an Intel Pentium® 4 processor: test programs automatically generated by the FRITS tool achieve 70% stuck-at fault cover-age for the entire chip, and when these programs are enhanced with manually generated tests, the fault coverage increases by 5%.

Differently from the previously described functional methods, in [3] the authors propose a two-steps methodology based on evolutionary algorithms: firstly a set of macros encrypting processor instructions is created, and in a second step an evolutionary optimizer is exploited to select macros and data values to conform the test program. The proposed approach is evaluated on a synthesized version of an 8051 microprocessor, achieving about 86% fault coverage. Later, in [2], a new version of the proposed approach is presented. The authors exploit a simulation-based method that makes use of a feedback evaluation to improve the quality of test programs: the approach is based on an evolutionary algorithm and it is capable of evolving small test programs that capture target corner cases for design validation purposes. The effectiveness of the approach is demonstrated by comparing it with a pure instruction randomizer, on a RTL description of the LEON2 processor. With respect to the purely random method, the proposed approach is able to seize three additional intricate corner cases while saturating the available addressed code coverage metrics. The developed validation programs are more effective and smaller in code size.

3 Proposed Approach

The previously described test generation cases show that evolutionary algorithms can effectively face real-world problems. However, when exploiting a low-level description of the processor under evaluation, simulation-based approaches require huge elaboration times.

We propose a methodology able to exploit a high-level description of a pipelined processor core in the generation process: the required generation time is thus reduced with respect to techniques that use a low-level description during the generation phase, such as the gate-level netlist, as reported in [10]. In the proposed approach, it must be noticed that the processor netlist is only used at the end of the generation process to assess the methodology results, performing a complete fault simulation.

The generation process is supported by the on-time automated generation of a FSM that models the excited parts of the processor core and drives the evolution process by indicating the unreached components on the processor core. In addition, we consider also high-level coverage metrics to improve the evolution.

It is possible to simply define a pipelined microprocessor as the interleaving of sequential elements (data, state and control registers), and combinational logic blocks. The inputs of the internal combinatory logic blocks are dependent on the instruction sequence that is executed by the processor and on the data that are processed.

One way to model a microprocessor is to represent it with a FSM. Coverage of all the possible transitions in the machine ensures thoroughly exercising the system functions. Additionally, the use of the FSM transition coverage has the additional advantage that it explicitly shows the interactions between different pipeline stages. Thus, we define the *state word* of a pipelined processor FSM model as the union of all logic values present in the sequential elements of the pipeline, excluding only the values strictly related to the data path. Consequently, the FSM transits to a new state at every clock cycle, because at least one bit in the state word is changed due to the whole interaction of the processor pipeline.

Figure 1 shows the proposed framework. The evolutionary core, called μGP^3 [15], is able to generate syntactically correct assembly programs by acquiring information about the processor under evaluation from an user-defined file called *Constraint Library*. When the process starts, the evolutionary core generates an initial set of random programs, or individuals, exploiting the information provided by the library of constraint. Then, these individuals are cultivated following the Darwinian concepts of natural evolution. Every test program is evaluated resorting to external tools that simulate the high level description of the processor core, resorting to a logic simulator at RTL, and generate a set of high-level measures. Contemporary, during the logic simulation, the FSM status is captured at every clock cycle, and for every evaluated test program the visited states and the traveled transitions are reported back to the evolutionary core as part of the evaluation of the goodness of an individual, called *fitness value*. The interaction between the different elements composing the fitness value guarantees good quality regarding the fault coverage against a specific fault model at gate level. Fitness values gathered during the logic simulation, for example code coverage metrics such as *Statement coverage* (SC), *Branch coverage* (BC), *Condition coverage* (CC), *Expression coverage* (EC), *Toggle coverage* (TC), are suitable for guiding the evolution of test programs. Simultaneously, maximizing the number of traversed transitions of the FSM model, assures a better result at gate level.

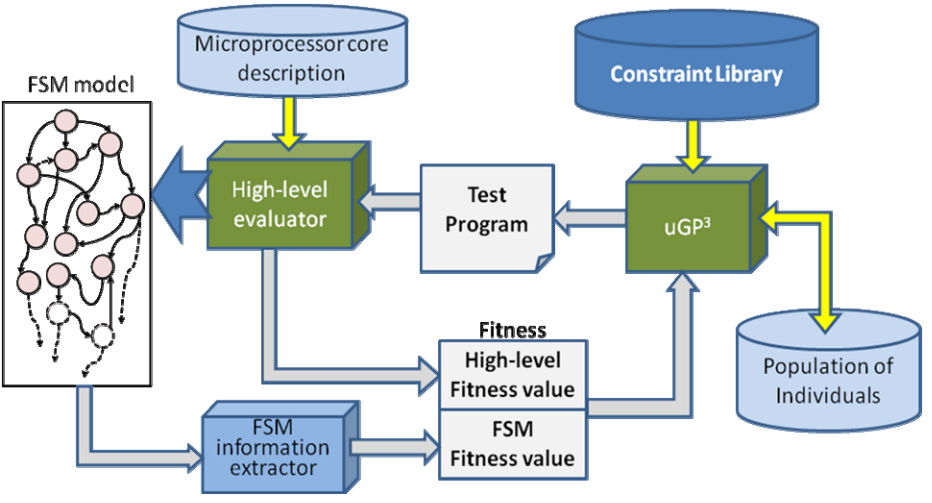

Fig. 1. Test generation framework.

Only the best individual is fault simulated in order to assess its fault coverage properties, reducing generation times. In the following paragraphs, we briefly describe in more detail the most significant elements present in the described framework.

3.1 µGP^3

µGP^3 represent individuals, in this case candidate test programs, as constrained tagged graphs; a tagged graph is a directed graph every element of which may own one or more tags, and that in addition has to respect a set of constraints. A tag is a name-value pair used to add semantic information to graphs, augmenting the nodes with a number of parameters, and also to uniquely identify each element during the evolution. Graphs are initially generated in a random fashion; subsequently, they may be modified by genetic operators, such as the classical mutation and recombination. The genotype of an individual is described by one or more constrained tagged graphs.

The purpose of the constraints is to limit the possible productions of the evolutionary tool, also providing them with semantic value. The constraints are provided through a user-defined library that supplies the genotype-phenotype mapping for the generated individuals, describes their possible structure and defines which values the existing parameters (if any) can assume. To increase the generality of the tool, constraint definition is left to the user.

In this specific case the constraints define three distinct sections in an individual: a program configuration part, a program execution part and a data part or stimuli set. The first two are composed of assembly code, the third is written as part of a VHDL testbench. Though syntactically different, the three parts are interdependent in order to obtain good solutions.

Individual fitness values are computed by means of one or more external evaluator tools. The fitness of an individual is represented by a sequence of floating point

numbers optionally followed by a comment string. This is currently used in a prioritized fashion: one fitness A is considered greater than another fitness B if the n-th component of A is greater than the n-th component of B and all previous components (if any) are equal; if all components are equal then the two fitnesses are considered equal.

The evolutionary tool is currently configured to cultivate all individuals in a single panmictic population. The population is ordered by fitness. Choice of the individuals for reproduction is performed by means of a tournament selection; the tournament size τ is also endogenous. The population size μ is set at the beginning of a run, and the tool employs a variation on the plus ($\mu+\lambda$) strategy: a configurable number λ of genetic operators are applied on the population. All new unique individuals are then evaluated, and the population resulting from the union of old and new individuals is sorted by decreasing fitness. Finally, only the first μ individuals are kept.

The possible termination conditions for the evolutionary run are: a target fitness value is achieved by the best individual; no fitness increase is registered for a predefined number of generations; a maximum number of generations is reached.

3.2 FSM Extractor

The proposed methodology is based on modeling the entire processor core as a FSM which is dynamically constructed during the test generation process. Thus, differently from other approaches, the FSM extraction is fully automated, and demands minimum human effort: the approach only requires the designer to identify the memory elements of the pipeline registers in the RTL processor description that will determine state characteristics of the FSM. The key point behind the FSM extractor is to guide the evolution trough a high-level model of the processor core that summarizes the capacity of excitation of the considered test program. The FSM information extractor receives from the external evaluator (e.g., a logic simulator) the activity of the pipeline registers of the processor core at every clock cycle, then, it computes for every clock cycle the processor state word and extracts the visited states and the traversed transitions.

Given the dynamic nature of the FSM construction, it is not possible to assume as known the maximum number of reachable states, not to mention the possible transitions. For this reason, it is impossible to determine the transition coverage with respect to the entire FSM.

The implemented evaluator, that includes the logic simulator and the FSM information extractor, collects the output of the simulation and dynamically explores the FSM; it assesses the quality of test program considering the transition coverage on the FSM and the code coverage metrics. The fitness fed back to the evolutionary tool is composed of many parts: the FSM transition coverage followed by all other high-level metrics (SC, BC, CC, EC, TC).

Let us consider the mechanisms related to hazard detection and forwarding activation in a pipelined processor: in order to thoroughly test them, it requires to stimulate the processor core with special sequences of strongly dependent instructions able to activate and propagate possible faults on these pipelined mechanisms. Facing this problem by hand requires a very good knowledge about the processor core to carefully craft a sequence of instructions able to actually excite the mentioned pipelined elements. Additionally, this process may involve a huge quantity of time.

On the other hand, state-of-the-art test programs usually do not target such pipeline mechanisms, since their main concern is exciting a targeted functional unit through carefully selected values, and not to activate the different forwarding paths and other mechanisms devoted to handle data dependency between instructions [4].

As a matter of fact, it is possible to state that a feedback based approach able to collect information about the interaction of the different instructions in a pipelined processor as the one described before, allows the evolution of sequences of dependent instructions that excite the mentioned pipeline mechanisms.

4 Case Study and Experimental Results

The effectiveness of the EA-based proposed methodology has been experimentally evaluated on a benchmark SoC that contains the OpenRISC processor core and some peripheral cores, such as the VGA interface, PS/2 interface, Audio interface, UART, Ethernet and JTAG Debug interface. The SoC uses a 32 bit WISHBONE bus rev. B for the communication between the cores. The operating frequency of the SoC is 150 MHz. The implemented SoC is based on a version publicly available at [14].

The OpenRISC processor is a 32 bit scalar RISC architecture with Harvard microarchitecture, 5 stages integer pipeline and virtual memory support. It includes supplementary functionalities, such as programmable interrupt controller, power management unit and high-resolution tick timer facility. The processor implements a 8Kbyte data cache and a 8Kbyte instruction cache 1-way direct mapped; the instruction cache is separated from the data cache because of the specifics of the Harvard microarchitecture.

In our experiments we decide to tackle specifically the processor integer unit (IU) that includes the whole processor pipeline. This unit is particularly complex and important in pipelined processors, since it is in charge of handling the flow of instructions elaborated in the processor core.

The pipelined processor is described by eight verilog files, counting about 4,500 lines of code. Table 1 describes some figures that are used to compute RTL code coverage and toggle metrics. Additionally, the final line shows the number of stuck-at faults (*S@ faults*) present in the synthesized version of the targeted module.

The state word is defined as the union of all memory elements composing the processor pipeline, excluding only the registers that contain data elements. Data registers are excluded because we are mainly interested in the control part of the pipeline, and not in the data path.

Table 1. IU information facts

OR1200 IU	
Lines	4,546
Statements	466
Branches	443
Condition	53
Expression	123
Toggle	3,184
S@ faults	13,248

Thus, considering the registers available in every stage of the processor pipeline, a state word contained 237 bits is saved at every clock cycle during the logic simulation of a test program allowing us to dynamically extract the processor FSM.

In order to monitor the elements contained in the state word of the pipeline at every clock cycle, we implemented a *Programming Language Interface* module, called *PLI*, that captures the information required during the logic simulation of the RTL processor. The PLI module is implemented in C language, counting about 200 lines of code. The module is compiled together with the RTL description of the processor core, exploiting a verilog wrapper.

Once a test program is simulated, a script *perl* extracts the information regarding the number of visited states as well as the number of traversed transitions obtained by the considered program. This information is collected together to the high-level coverage metrics provided by the logic simulator and the complete set of values is fed back to the evolutionary engine in the form of fitness value of the test program.

The configuration files for the evolutionary optimizer are prepared in XML and count about 1,000 lines of code. Finally, additional *perl* scripts are devised to close the generation loop.

A complete experiment targeting the OR1200 pipeline requires about 5 days. At the end of the experiment, an individual counting 3,994 assembly lines that almost saturate the high level metrics is created; the same individual obtains about 92% fault coverage against the targeted fault model.

Compared to manual approaches reported in [4], that achieve about 90% fault coverage in the considered module, the results obtained in this paper improve the fault coverage by about 2%, and can be thus considered promising.

5 Conclusions

Tackling microprocessor testing with evolutionary algorithms proved effective in many works in literature, but this methodologies share a common disadvantage: the time needed to evolve a suitable test program is considerable.

In order to solve this problem, a novel evolutionary-based test approach is proposed. The approach exploits a high-level description of the device under test, along with a dynamically built FSM model, to esteem the fault coverage of the candidate test programs. Thus, a reliable evaluation of the goodness of the programs is obtained without resorting to time-expensive simulations on low-level models.

The proposed framework is assessed on a OpenRISC processor. Experimental results show a total of 92% fault coverage against the targeted fault model.

Acknowledgements. We would like to thank Danilo Ravotto and Xi Meng for technical assistance during experimental evaluation.

References

[1] Bushard, L., Chelstrom, N., Ferguson, S., Keller, B.: DFT of the Cell Processor and its Impact on EDA Test Software. In: IEEE Asian Test Symposium, pp. 369–374 (2006)

[2] Corno, F., Sanchez, E., Sonza Reorda, M., Squillero, G.: Automatic Test Program Generation – a Case Study. IEEE Design & Test of Computers 21(2), 102–109 (2004)

[3] Corno, F., Sonza Reorda, M., Squillero, G., Violante, M.: On the Test of Microprocessor IP Cores. In: DATE, pp. 209–213 (2001)
[4] Gizopoulos, D., Psarakis, M., Hatzimihail, M., Maniatakos, M., Paschalis, A., Raghunathan, A., Ravi, S.: Systematic Software-Based Self-Test for pipelined processors. IEEE Transactions on Very Large Scale Integration (VLSI) 16(11), 1441–1453 (2008)
[5] Mak, T.M., Krstic, A., Cheng, K.-T., Wang, L.-C.: New challenges in delay testing of nanometer, multigigahertz designs. IEEE Design & Test of Computers 21(3), 241–248 (2004)
[6] May, G., Spanos, C.: Fundamentals of Semiconductor Manufacturing and Process Control, p. 428. Wiley-IEEE Press publisher (2006) ISBN: 9780471790280
[7] Parvathala, P., Maneparambil, K., Lindsay, W.: FRITS – A Microprocessor Functional BIST Method. In: IEEE Intl. Test Conf., pp. 590–598 (2002)
[8] Pradhan, D.K., Harris, I.G.: Practical Design Verification. Cambridge University Press, Cambridge (2009) ISBN: 9780521859721
[9] Psarakis, M., Gizopoulos, D., Sanchez, E., Sonza Reorda, M.: Microprocessor Software-Based Self-Testing. IEEE Design & Test of Computers 27(3), 4–19 (2010)
[10] Sanchez, E., Sonza Reorda, M., Squillero, G.: Test Program Generation From High-level Microprocessor Descriptions. In: Test and Validation of Hardware/Software Systems Starting from System-level Descriptions, 179 p. Springer, Heidelberg (2005) ISBN: 1-85233-899-7, pp. 83-106
[11] Shen, J., Abraham, J.: Native mode functional test generation for processors with applications to self-test and design validation. In: Proceedings IEEE Intl. Test Conf., pp. 990–999 (1998)
[12] Speek, H., Kerchoff, H.G., Sachdev, M., Shashaani, M.: Bridging the Testing Speed Gap: Design for Delay Testability. In: IEEE European Test Workshop, pp. 3–8 (2000)
[13] Wang, S., Gupta, S.K.: ATPG for heat dissipation minimization during scan testing. In: ACM IEEE Design Automation Conference, pp. 614–619 (1997)
[14] http://www.opencores.org/
[15] http://ugp3.sourceforge.net/