On the Test of Microprocessor IP Cores

F. Corno, M. Sonza Reorda, G. Squillero, M. Violante

Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
http://www.cad.polito.it/

Abstract

Testing is a crucial issue in SOC development and production process. A popular solution for SOCs that include microprocessor cores is based on making them execute a test program. Thus, implementing a very attracting BIST solution. This paper describes a method for the generation of effective programs for the self-test of a processor. The method can be partially automated, and combines ideas from traditional functional approaches and from the ATPG field. We assess the feasibility and effectiveness of the method by applying it to a 8051 core.

1 Introduction

Technology advances made it possible to integrate on a single chip enormous numbers of transistors, thus allowing the inclusion on a single chip of entire systems, including microprocessors, ASICs, memories, and peripherals. The development of this new class of systems, called Systems-On-a-Chip (or SOCs), is made easier by the availability of Intellectual Property (IP) cores provided by third parties. By exploiting these cores, ASIC designers easily and quickly include into their SOCs complex cells such as microprocessors, peripherals, or dedicated blocks (e.g., MPEG encoders-decoders). Testing SOCs is rapidly becoming a major issue, mainly because of the complexity of the blocks they embed and of their limited accessibility. Other problems, such as the time required for testing and the power consumed during this phase, also contribute to make very challenging the problem of SOC testing.

In this paper we tackle test of microprocessor and microcontroller IP cores that are often embedded in SOC designs. Popular solutions are based on adopting a full-scan approach, or on forcing the core to execute a given test program. However, the first solution has several drawbacks, mainly in terms of test length (and required size of the ATE memory) and performance degradation (due to the scan chain insertion). Moreover, it does not allow an at-speed test of the core and requires special chip-level architectures for allowing external access to the scan chains. In this paper we only deal with the second solution that, in principle, does not suffer of the above limitations and allows test execution at the processor speed with no area overhead. This approach has some requirements: first, some memory space must be available to store the test program; second, a way should be devised to observe the results of the test program execution; finally, the test program itself must be available. Fault coverage is expected to be lower than in the full-scan approach, but suitable test programs can attain acceptable coverage levels.

In this paper we propose a test solution partly derived from the one described in [4]: we assume that a RAM memory of sufficient size is available on the SOC, and that this memory can be easily accessed from both the external ATE and from the microprocessor core. In this way, the 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. A suitable solution for observing the results of the test program execution has also been devised, based on repeatedly activating an ad hoc software procedure which is in charge of compacting the values produced by the test program and of writing back the resulting signature in memory, so that it can be easily read back and observed by the ATE. Finally in this paper, we address the issue of generating a test program that matches the above assumptions and is able to effectively test the processor core, and propose a method to generate such program.

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 [1]. 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 [2] 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 [3] that is based on generating random sequences of instructions. However, this approach also 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 [6]. On the other hand, Utamaphethai et al. showed a method for generating instruction sequences for validating the branch prediction mechanism of the PowerPC604 [7]. Generated sequences are very effective, but the methodology exploits a deep knowledge of the processor and it cannot be easily applied on general designs.

The new approach proposed in this paper is particularly suited when a netlist of the processor is available (although possibly encrypted). This situation often arises then the microprocessor is bought from third parties, and either a test program is not available, or the fault coverage it attains is not sufficient. The proposed method requires a limited amount of manual work aimed at developing a library of *macros*, that are able to excite all the functions of the processor. A macro is associated to each machine-level instruction; each macro is composed of few instructions, aimed at activating the target instruction with some 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 [1]; 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 [1]. The final test program is composed of a proper sequence of macros taken from this library, each activated with proper values for its parameters. The choice of the most suitable parameters value is accomplished by resorting to a Genetic Algorithm. Experimental results supporting the effectiveness of the method are reported for a core implementing the Intel 8051 microcontroller.

The paper is organized as follows. Section 3 outlines the assumptions we made in terms of test strategy and SOC architecture for test. Section 0 presents an overview of the test program generation approach we propose, as well as details on the macros and on the Genetic Algorithm we adopted. Section 4 reports some preliminary experimental results assessing the effectiveness of our approach, while Section 5 draws some conclusions.

2 Test Strategy and Test Architecture Assumptions

For the purpose of this work we considered a SOC design including a microprocessor or microcontroller IP core whose test is not performed resorting to any Design for Testability mechanism, such as partial- or full-scan. The existence of some BIST structure for testing internal parts of the core (e.g., the control memory) is compatible with our approach, provided that the BIST structure can be autonomously activated from the outside.

We assume that the processor core is able to access a RAM memory that must also be accessible from the outside. Note that this accessibility can be limited in terms of speed and flexibility, since we only require that the ATE is able to load this memory with the test program for the processor core, and to read the compacted output produced by the program. Therefore, even a low-cost ATE can be explited for this purpose. In [4] it is suggested that the transfer of data between the ATE and the memory can be implemented resorting to a DMA mechanism. Moreover, the test program can possibly be transferred to memory in a compacted form. In this case, the processor is in charge of de-compacting the test program before its execution. A possible architecture of the considered SOC design is shown in Fig. 1. Alternatively, the test program could be stored in a ROM, thus allowing the activation of the whole test process in a completely autonomous way, as it is required in an on-line test mechanism.

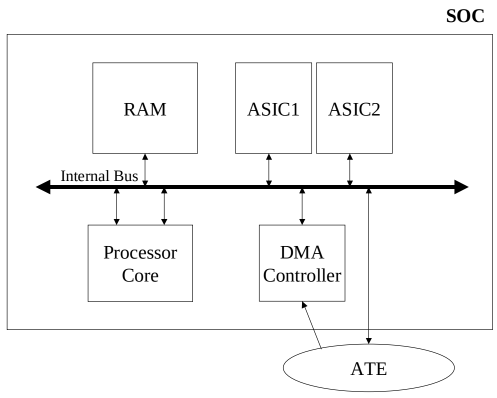

Fig. 1: SOC design architecture.

The test program is composed of a sequence of instructions, whose execution causes the activation and propagation of faults inside the processor core. In order to make the effects of possible faults observable, the results produced by the program are written in one or few cells in the RAM, named *result words*. After the execution of each macro, the content of each result word is immediately processed by an ad hoc procedure, which compacts the sequence of values written on each of them, and produces a signature which is made available to the ATE, e.g., by writing it into a specified position in the memory.

3 Test Program Generation

3.1 Overview

Generating a test sequence for a microprocessor is a challenging task, usually much harder than for a generic ASIC. In fact, the internal structure of a microprocessor and its behavior make unfeasible the adoption of a standard ATPG. In particular, the internal structure is based on a sequentially complex *decode and control unit* that decodes instructions and sends the appropriate control signals to a large *data-path*, which may include hard-to-test elements such as multipliers and dividers. The two main components of a microprocessor require different approaches for their test: the decode and control unit can be seen as a complex Finite State Machine (FSM), and its test requires suitable sequences to traverse its state graph to excite and observe possible faults. Therefore, its test is mainly a sequential problem. On the other side, the complexity of testing the data-path mainly stems from the complexity of some combinational (or slightly sequential) parts, such as multipliers and dividers, and requires a careful selection of operands to be written on the inputs of these parts.

Finally, 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.

As a consequence of the above considerations, the approach we propose is based on two steps.

- First, we build a set of macros, each composed of 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). Each macro owns several parameters, corresponding to the operands of the instructions it is composed of. The macros are stored in a library, which is exploited in the second step.
- The second step is a search algorithm and aims at selecting from the library a sequence of macros, and at choosing the values for their parameters, such that maximum Fault Coverage can be reached. The first goal is attained by adaptively selecting macros based on the experience gathered during previous activations; a Genetic Algorithm implements the latter.

The pseudo-code of the search algorithm we propose is reported in Fig. 2. At each iteration, a macro is randomly selected from the library resorting to a probability distribution evolving during the process. In order to implement the procedure `select_the_operands`, a Genetic Algorithm is then activated, whose goal is to find the values for the macro operands that maximize the number of faults detected by the macro. If the algorithm is successful, and at least a new fault is detected, the macro is added to the final test program; otherwise, the macro probability of being selected is decreased, thus reducing its chance of being selected again in the future. The stopping condition is true when either the Fault Coverage reaches a given threshold, or the maximum number of iterations has been reached.

```
do
{   m = select_a_macro();
    O = select_the_operands(m);
    F = compute_detected_faults(m, O);
    if( F is not empty )
        add m(O) to the test program;
} while( stopping_condition() == FALSE );
```

Fig. 2: pseudo-code of the algorithm.

3.2 Macros

The macro library is built following some guidelines:

- for each machine-level instruction in the processor instruction set (hereinafter called the target instruction) one macro is inserted in the library;
- each macro aims at activating the target instruction, and at verifying the correctness of the instruction execution by propagating the outputs to one or more result words (see Section 2).

As an example, the macro for any arithmetic instruction is organized in three phases:

1. *load* values in the two operands of the instruction; according to the machine-level instruction we are considering, these parameters exploit different addressing modes (register, immediate, memory, etc.); the actual value of each parameter for every macro activation will be defined during the optimization phase;
2. *execute* the instruction;
3. make the result *observable* by writing it (directly or indirectly) to one or more result words.

The reader should note that the purpose of the macro is to make observable the complete result of each instruction, which also includes any flag that is possibly affected by the instruction itself. As an example, Fig. 3 reports the code (for sake of readability we use a pseudo-8086 assembly language) for the macro concerning the addition instruction between registers; `K1` and `K2` are two parameters, whose value (in terms of addressing mode or immediate value) determines the Fault Coverage the macro attains. `RW` and RW2 are result words in the memory: just after every call to the macro, a compaction procedure is activated, reading RW and RW2 and updating the signature, which is then observed from the outside at the end of the test program.

All macros for arithmetic and logical instructions follow the same template reported in Fig. 3.

```
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 reg to RW
```

Fig. 3: pseudo-code of the macro for the ADD instruction.

A special class of macros is built to test conditional and unconditional branch instructions: in this case the result produced by the instruction is the execution of one instruction flow or another. Therefore, different values are written to a result word in the two cases, allowing the detection of faults in both the circuitry evaluating the condition, and in that modifying the Program Counter. When conditional instructions are considered, phase 1 builds the condition, phase 2 evaluates the condition and possibly performs the branch, and phase 3 writes different values depending on the result of condition evaluation. Fig. 4 shows a sample macro addressing the JG (Jump on Greater) instruction: depending on the value appearing on the result word RW one can detect possible faults affecting the circuitry evaluating the condition and performing the jump.

```
MOV BX, 0       ; clear BX
CMP AX, K1      ; compare reg AX with K1
JG  IS_ABOVE    ; jump if AX > K1
MOV BX, 1       ; move 1 to BX
    IS_ABOVE:
MOV RW, BX      ; write BX to RW
```

Fig. 4: pseudo-code of the macro for the JG instruction.

3.3 Operand Selection

Once a macro has been selected from the library, a Genetic Algorithm is activated to identify the best values for its parameters. By suitably selecting these values, the algorithm chooses the values for immediate operands, or those to be written in the registers or memory cells used by the target instruction.

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

- chromosomes are bit strings corresponding to the concatenated operands; their length changes depending on the macro;
- the standard random mutation operator has been adopted, which randomly selects a bit in the chromosome, and complements it;
- the one-cut cross-over operator has been adopted;
- chromosomes are selected for mating using a linearized fitness function and a roulette wheel mechanism;
- population management is performed using elitism;
- the algorithm is stopped when a steady state is reached, i.e., when a given number of generations have elapsed without detecting new faults.

The fitness function of a chromosome is the number of faults detected by the macro when it is fault simulated with the operand values given by chromosome on a fault list composed of gate-level stuck-at faults.

4 Experimental Results

In order to practically assess the effectiveness of the proposed approach we implemented an Automatic Test Program Generator System (ATPGS) whose architecture is sketched in Fig. 5. The ATPG system amounts to about 1,000 lines of C code that interact with an in-house developed Fault Simulator (about 3,000 lines of code) based on a fault-parallel event-driven PROOFS-like [5] algorithm for obtaining information about the effectiveness of each set of macro parameter values.

The system has been evaluated on a description of the Intel 8051 microcontroller. The Register Transfer level model of the 8051 (which amounts to 5,800 lines of VHDL code) has been synthesized using Synopsys `design_analyzer` and mapped on a standard library. The resulting netlist amounts to about 6,000 gates, not including the internal memory for code. The fault simulator is able to simulate the entire 8051 while it executes the test program from the memory. Stuck-at faults are injected in the combinational and sequential logic, excluding memories. However, fault propagation across memory cells is correctly taken into account.

We defined a library composed of 213 macros (the number of instruction in each macro ranges from 3 to 6) and then we run our Automatic Test Program Generator. Macro generation took 2 working days of an experienced assembler programmer. The experiment has been performed on a Sun Enterprise 250 running at 400 MHz and equipped with 2 Gbytes of RAM; it required 24 hours of CPU time for both test program generation and processor Fault Simulation.

The values we used for the Genetic Algorithm parameters are reported in Tab. 1.

The results we obtained are reported in Tab. 2. To assess the effectiveness of the approach we propose, we compared it with the fault coverage obtained by a purely random approach. We generated a random test program, composed of randomly generated macros. The number of macros in the randomly generated program corresponds to the number of macros we simulated during the whole optimization process [Simulated]. Only macros that detected additional faults are included in the test set, and contribute to the test length [Final TS]. The program generated by an ATPG was able to identify more useful macros than the random approach, that comparatively wasted more simulation time for achieving lower fault coverage.

Parameter	**Value**
Number of individuals in the population	5
Number of new individuals at each generation	3
Number of generations	50
Mutation probability	(Chromosome length)$^{-1}$

Tab. 1: Genetic Algorithm Parameters

	Simulated [# Macros]	**Final TS [# Instr]**	**FC [%]**
ATPGS	7,000	624	85.19
Random	7,000	103	80.19

Tab. 2: Experimental Results

These preliminary results suggest that the approach is effective since it provides high Fault Coverage figures with respect to a purely random approach at a cost of an acceptable amount of CPU time; moreover, it does not require an in-depth knowledge of the processor under test (while functional approaches do) and it does not require additional test hardware to be added to the processor.

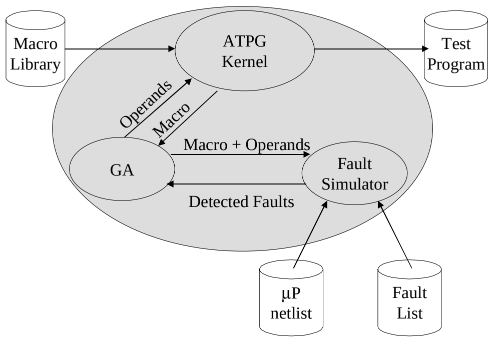

Fig. 5: the Automatic Test Program Generator System.

5 Conclusions

In this paper we presented an approach for the test of microprocessor and microcontroller test cores in a SOC. An approach to their test is proposed, which is based on first loading a test program for the core in an internal memory, and then letting the core to execute it. This approach implements a software BIST: it does not require any high-cost ATE equipment, and allows an at-speed test of the processor core. A method for computing the test program is also presented. The method requires a limited knowledge of the processor under test and is based on two steps. A programmer on the basis of the processor instruction set defines a set of macros. Then, a search algorithm selects a suitable set of macros that maximizes the attained Fault Coverage. In the selection process a Genetic Algorithm is exploited in order to defines the values for each macro parameters.

Preliminary results gathered on an Intel 8051 are provided. They show the feasibility of the method, and prove that the generated test program outperforms randomly generated programs at a cost of an acceptable amount of CPU time. The human effort in developing the information necessary for activating the ATPG method is much less than comparable approaches.

Currently we are experimenting different heuristics and fitness functions to enhance the search process. Moreover, we developed a new fault simulator, enabling the genetic algorithm to exploits a tighter interaction. Preliminary results are encouraging.

6 References

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

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

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

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

[5] 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

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

[7] 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