Effective Software Self-Test Methodology for Processor Cores

N. Kranitis, A. Paschalis
Department of Informatics & Telecommunications
University of Athens, Greece
{nkran | paschali}@di.uoa.gr

D. Gizopoulos
Department of Informatics
University of Piraeus, Greece
dgizop@unipi.gr

Y. Zorian
LogicVision,
San Jose, CA, USA
zorian@logicvision.com

Abstract

Software self-testing for embedded processor cores based on their instruction set, is a topic of increasing interest since it provides an excellent test resource partitioning technique for sharing the testing task of complex Systems-on-Chip (SoC) between slow, inexpensive testers and embedded code stored in memory cores of the SoC. We introduce an efficient methodology for processor cores self-testing which requires knowledge of their instruction set and Register Transfer (RT) level description. Compared with functional testing methodologies proposed in the past, our methodology is more efficient in terms of fault coverage, test code size and test application time. Compared with recent software based structural testing methodologies for processor cores, our methodology is superior in terms of test development effort and has significantly smaller code size and memory requirements, while virtually the same fault coverage is achieved with an order of magnitude smaller test application time.

1. Introduction

Almost every complex System-on-Chip (SoC) consists of at least one embedded processor core which may be either a general purpose processor or a special purpose processor for graphics, audio/video applications etc. with enhanced DSP functionality. Such processor cores are surrounded by memory cores (RAM or ROM) of various sizes used for code and data storage.

The complexity of SoC designs consisting of deeply embedded cores with poor accessibility makes their testing process a difficult task. Additionally, the increasing gap between the operating frequencies of Automatic Test Equipment (ATE) and the operating frequencies of SoC lead to the escape of failures that may be detected only when testing is performed in the actual speed of the IC (at-speed testing). The transfer of the SoC test task from an external ATE to an internal built-in self-test (BIST) mechanism provides significant advantages not only for processor cores but also for other types of cores like memories. The use of self-test methodologies for processor testing reduces yield loss and drives down the overall test cost of the SoC [1] while actual at-speed testing is possible. In addition, the use of self-test reduces the design cycle and thus improves time-to-market, while it also provides better Intellectual Property (IP) protection than classical scan-based external testing techniques.

When a self-test methodology is based on hardware mechanisms, special parts of the circuit are synthesized for Test Pattern Generation and Output Data Evaluation. In this case, the extra circuit area may be significant, but the most important is the possibility of significant performance loss due to the introduction of extra logic in the critical paths of the circuit. Recent applications of hardware-based commercial logic BIST techniques in large industrial designs and microprocessors [2], [3], [4], revealed that extensive design changes have to be performed (most of them manually). Theses changes have a negative impact in the circuit area and performance since extensive test point insertion is necessary to achieve an acceptable fault coverage mostly because of the random pattern resistance of the circuits. In the case of processor cores, which are very carefully designed IP entities optimized for performance or power consumption, such hardware-based self-test mechanisms that seriously impact performance and power consumption can be considered of limited practical value.

Software-based self-test methodologies for embedded processor cores have the advantage that they utilize the processor functionality and instruction set for both Test Pattern Generation and Output Data Evaluation and thus do not add hardware or performance overheads in the optimized design. Such self-testing approaches have been recently proposed in the literature, [4]-[8].

In [4] the concept of self-test signatures is introduced and a *structural* testing methodology for processor cores is presented. At the test preparation stage, pseudorandom patterns are used for each processor component in an iterative method taking into consideration the constraints imposed by its instruction set, based on the knowledge of the gate-level netlist of every component. At the test execution (test application) stage, pseudorandom test patterns developed at the test preparation stage and encapsulated into self-test signatures, are first expanded on-chip by a software-emulated LFSR (test generation program) and stored in embedded memory. Then, the pseudorandom test patterns are applied by software test application programs and responses are collected into memory again. At the test preparation stage, as an alternative, gate level ATPG can be used to generate test patterns for processor components in the iterative

constrained test generation method. This methodology is restricted by the need of gate-level details of the processor structure. Such information may not be available, but even in the case that it is actually available, the instruction set imposed constraint test generation of deeply embedded functional modules of the processor is a very time consuming task, which may not always lead to an acceptable fault coverage. Besides, the pseudorandom nature of the methodology leads to large self-test code, large memory requirements and excessive test application time (total number of processor clock cycles for the self-test session).

The processor self-testing approaches of [5], [6] rely on the use of random instruction sequences, while those of [7], [8] rely on pseudorandom operations and operands. In particular, the *functional* testing methodologies of [5] and [8] are based only on the knowledge of the instruction set architecture of the processor (instructions, registers, addressing modes). The obtained structural fault coverage is low due to the high level of abstraction of the methodology. Besides, the self-test code sequences and the test application time are both excessively large due to the use of a pseudorandom strategy. The large size of the self-test code significantly increases the overall test time since the self-test code sequences are downloaded into memory at the low frequency of the external tester.

An attractive alternative to pseudorandom software self-testing is *deterministic* software self-testing. As mentioned above, if the gate-level information of the processor core is available, deterministic patterns can be generated by an ATPG taking into consideration the control signal constraints imposed by the instruction set. These test patterns do not have an inherent regularity and thus are difficult to be generated by efficient software code based on small loops of instructions and compact test routines. Only in the case that the number of test patterns is very low this method is efficient. In this case the test patterns have to be downloaded by the external ATE into memory along with a simple test application program.

Fortunately, it has been shown that very small deterministic self-test sequences are sufficient for testing the functional modules of a processor datapath. The self-test sequences can be generated by simple hardware machines [9]-[12]. In the case of processor cores when no hardware additions may be affordable, the test sequences can also be generated by software routines [13]. When self-test routines are based on deterministic test sets for the functional modules, they have two main advantages. First, the self-test code is very small and based on short loops of a few instructions. Second, the achieved fault coverage is guaranteed for any width of the operands and any internal implementation of the functional modules [9]-[12].

In this paper, we propose a software-based self-test methodology for embedded processor cores that is based on the knowledge of the Instruction Set Architecture (ISA) of the processor and its *Register Transfer (RT) level* description. The RT level description showing the connections among the functional parts of the processor (ALUs, adders, multipliers, shifters, etc), the storage elements (registers, register files, flags) and the steering logic modules (multiplexers, bus elements) is a piece of information which is usually available and is much more easily managed than a detailed gate-level netlist. Therefore, a limited engineering effort is required. Our methodology is based on the application of *deterministic* test patterns targeting structural faults of individual processor components. The deterministic test patterns are not ATPG generated but are developed by our methodology in order to excite the entire set of operations that each component performs. For each component operation (arithmetic or logic operations, data transfer operations etc) a basic self-test routine is developed based on a deterministic test set for the component (i.e. functional modules like an ALU, a shifter or a multiplier) that performs the operation after mapping each operation to a processor instruction. The derived self-test code is compact due to the use of small regular test sets. The regularity of the basic test sets for the functional module components is essential for the success of the proposed software self-test methodology, since it is the driving force for the small size of the self-test code and thus its small memory requirements.

We demonstrate our methodology in detail using the same processor core used in [4] and provide detailed experimental results. The first advantage of our methodology is the *significantly smaller* self-test code size and memory requirements. The self-test code reduction leads to an important reduction in the time required to download the program from external tester into internal memory. Besides, the number of *processor clock cycles* required for execution of our test code is by *an order of magnitude* smaller than [4]. The combination of the reduced test code size and reduced test code execution time leads to a large total reduction in the overall test application time for the processor.

2. Self-Test Methodology

2.1 Overview

According to the proposed self-test methodology for processor cores, self-test routines based on the instruction set architecture and the RT level description of the processor core are first developed. These routines are either stored in a ROM for in-the-field periodic testing or are loaded in a RAM by a low cost external ATE for manufacturing testing.

Subsequently, these self-test routines are executed at-speed to generate the necessary test patterns for testing the complete set of operations performed by the components of the processor and test results are stored back in RAM. These test results may be in a compacted or uncompacted form and can be again unloaded by the low cost external ATE for manufacturing testing.

The proposed processor self-test methodology due to its compact test code and reduced data is also an excellent solution for in-the-field periodic testing where the small size of memory portions dedicated to test is important.

2.2 Testability of processor components

Every processor component is either a functional unit, an internal processor register or a steering logic element (i.e. multiplexer, tri-state buffer).

The testability of the processor components depends on the controllability of the component inputs and the observability of the component outputs. If the component is a functional unit or steering logic element, its testability depends directly on the controllability of the processor registers that drive the component inputs and the observability of the processor registers driven by the component outputs. The *data* inputs of a functional unit component are usually fully controllable and data outputs are fully observable since there are several registers (i.e. the accumulator, general purpose registers) accessible through primary inputs and outputs, respectively, using processor instructions. On the other hand, the *control* inputs and outputs of a functional unit component are less controllable and thus, special self-test routines have to be developed. This testing difficulty results from the fact that these control inputs and outputs are driven and drive, respectively, hard to control and observe processor registers like the status register.

In the case that the component is an internal processor register, its testability depends on the existence of processor instructions (or instruction sequences, in general) capable of assigning the required test patterns to them (from primary inputs or on-chip generated) and propagate their values to primary outputs (test response). Therefore, each processor register has its own controllability and observability characteristics.

As in previous processor self–test approaches we do not develop a special routine for the control unit component of the processor. This component is tested sufficiently when other specific processor components are tested with their self-test routines.

2.3 Self-Test Program Development

The development of the dedicated *self-test routines* for testing specific processor components according to our test methodology is performed in three steps:

Information extraction (Step 1). From the processor instruction set architecture and RT level description, we extract the effects of the execution of each instruction to every component. Thus, for every instruction I and for every component C participating in the instruction execution the following information is extracted:

- The specific operation O, that the component performs along with the relevant control signals which are enabled by the control unit for the execution of this operation.
- For every operation O, the involved internal processor registers and memory for data and control storage, along with their controllability and observability characteristics.

The extracted pieces of information from this step are used for (a) mapping the component operations and related control signals to processor instructions, (b) sorting the instructions according to the controllability and observability characteristics of the involved processor registers and memory.

Instruction selection (Step 2). For every component C we first identify the set of operations O_C that component C performs.

We denote $I_{C,O}$ the set of processor instructions that, during execution, enable the *same* control signals and cause, component C to perform operation O.

It is evident that for each component C there is at least one processor instruction that, during its execution, causes component C to perform operation O, i.e. $I_{C,O} \neq \varnothing$. The instructions which belong to the same set $I_{C,O}$:

- have different observability properties since, when operation O is performed, the outputs of component C drive internal processor registers with different observability characteristics.
- have different controllability properties since, when operation O is performed, the inputs of component C are driven by internal processor registers with different controllability characteristics

After identification of the set $I_{C,O}$ for every component operation we select an instruction I of the set $I_{C,O}$ according to the following criteria:

Criterion 1: Discard instructions belonging to $I_{C,O}$ that, when operation O is performed, the outputs of component C do not propagate to an internal processor register. This criterion is a simple one since if the effect of an instruction to a component operation is not stored in a register the possibly faulty component output cannot be propagated furthermore.

The remaining instructions belonging to $I_{C,O}$ are sorted in an *Instruction Priority List (IPL)* using the extracted information of Step 1 in accordance to the following criteria:

Criterion 2: Between instructions I_A and I_B belonging to $I_{C,O}$, I_A is ranked higher than I_B in the IPL (higher priority), if it requires a smaller instruction sequence to propagate the outputs of component C through the related internal processor register to primary output ports. That means that instruction I_A is more easily observable than I_B and should be preferred over I_B.

Criterion 3: If Criterion 2 ranks two different instructions I_A and I_B belonging to $I_{C,O}$, at the same position in the priority list, we select the one that requires smaller instruction sequence to generate a specific test pattern at the internal processor register that drives the inputs of component C. This instruction sequence can be either a simple load instruction (for a single test vector that comes from primary input, that is stored in memory), or generated on chip by an efficient deterministic test generation algorithm (for complex arithmetic and logic operations), but in both cases the vector should end in the internal processor register that drives the inputs of component C.

The above three criteria aim to the efficiency of self-test routines for *single components.* If a global optimization is performed for the total test program, different criteria could be followed.

Operand selection (Step 3). In this step we consider the deterministic operands that must be applied to each component to achieve high structural fault coverage. Each component category (functional modules, registers, steering logic) requires special test patterns.

The most difficult to test processor components are the functional module components. The functional units (i.e. ALU) data inputs and outputs are usually fully accessible, since their inputs are driven directly from memory or very well controllable processor registers (i.e. accumulator, general purpose registers) and their outputs drive well observable processor registers or directly propagate to primary outputs. Thus, after applying each test and the test response is kept in a very well observable register, the most suitable processor instruction for propagating test response to primary output (memory) is a simple Reg→Mem instruction like STORE.

If part of the test response of a component is not driven to a well accessible internal register (that is the case of flag outputs driving status register) an extra instruction sequence is required to propagate first to accessible registers and then to primary outputs (memory).

For hard to test functional modules, self test routines are developed based on deterministic test sets providing a test code library. These software test routines, developed for components implementing a large number of arithmetic and logic operations (i.e. ADD, MULTIPLY, AND), are generated on-chip using the processor instruction set and consist of efficient loops. For this reason they are very compact and require a very small number of bytes. The operations performed in these loops are very simple operations (mostly additions) on contrast with a software LFSR implementation in a pseudorandom based methodology (where complex parity bit computations, bit wise logic operations and shifting must be performed). They also apply a limited number of test vectors resulting in very short test program length.

The overall process is outlined in the following:

```
for (each component C)  {
  for (every operation o∈Oc) {
       Determine Ic,o
       Select I ∈ Ic,o, using
             controllability and
             observabillity criteria
       Using instruction I,
             apply deterministic data
             patterns at C data inputs
   }
  if fault coverage ≥ target, exit
}
```

One should note that in a processor instruction set, it is very usual for two different components C1, C2 that:

$$\exists\, I : I_{C1,O1} \cap I_{C2,O2} \neq \varnothing$$

That means that there are instructions their execution applies to two different components C1, C2 operations O1,O2. If the outputs of both components are propagated to primary outputs, although test vectors were prepared for one of the two (i.e, C1), faults of C2 will be also detected during the instruction execution and output propagation. This results to the fact that during software testing of the hard to test components, several other processor components are tested as well, thus eliminate the necessity for considering every individual component in the test preparation process. Therefore, processor components with a high potential to activate multiple operations in other components during the application of instructions that test them must be considered first. Such components are the functional modules and as it is demonstrated in the next section, only the two functional modules of the example processor are sufficient to provide a very high fault coverage for all the others.

An important aspect of the proposed methodology is that it does not require synthesis and gate level description of the tested components. Due to its deterministic nature, a basic knowledge of information in terms of functionality or functional blocks inside the component is required. This information is easily implied by the control signal applied to it or a related output. The only information necessary is an RT level description of the processor.

3. Experimental results

We have evaluated the effectiveness of the proposed methodology on the same accumulator-based processor core Parwan (see Figure 1) as in [4]. It is an 8-bit CPU with a 12-bit address bus (4Kbytes memory). Parwan instruction set includes most common instructions like load and store, arithmetic and logical operations, jump and branch instructions. It also supports direct and indirect addressing modes. Considering both addressing modes, Parwan instruction set has a total of 24 different instructions.

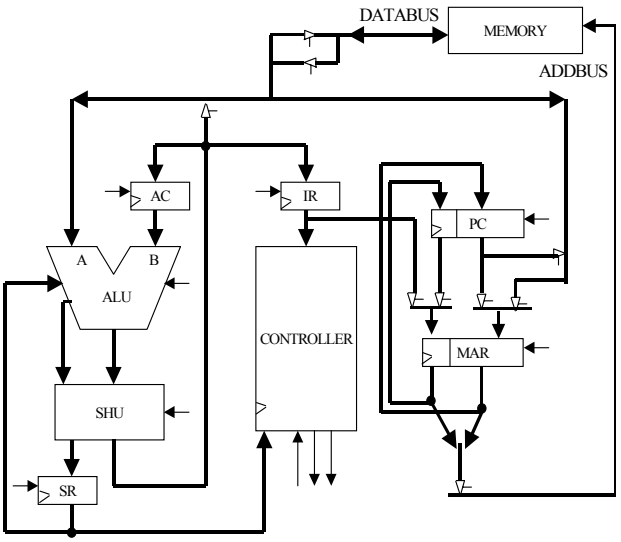

Figure 1. Parwan CPU

Parwan includes the following components: arithmetic logic unit (ALU), shifter unit (SHU), accumulator (AC), program counter (PC), status register (SR), memory address register (MAR), instruction register (IR) and control unit (CTRL). From the seven processor components only ALU and SHU are combinational circuits and also the only functional units. These two components have fully controllable and observable data inputs and outputs as a result of the fact that they operate in operands from/to memory and accumulator which is fully accessible in terms of controllability and observability.

The Parwan processor synthesized design contains a total of 1300 2-input NAND gate equivalents out of which 53 are flip-flops. Tristate buffers control the 8-bit data and 12-bit address buses. For synthesis, VHDL simulation and fault simulation we used the Leonardo, ModelSim and FlexTest products, respectively, from Mentor Graphics in a test evaluation framework similar to [4].

Test programs were prepared for the ALU and SHU components according to our methodology. A total of 36 deterministic test patterns for the six ALU operations and 25 test patterns for the three SHU operations are applied respectively to ALU and SHU, in the form of loops consisting of simple instructions. During test program execution, other components including the control unit are tested as well.

Test program statistics such as the number of instructions, the program size in bytes, the response data size in bytes and the program execution time in clock cycles are presented in Table 1. In the same Table, we provide statistics from the software self test methodology of [4].

	Method of [4]	***Proposed Method***	***Reduction***
# instructions	575	444	22.8%
Program Size (bytes)	1,129	885	21.6%
Response data (bytes)	514	122	76.3%
Exec. Time (cycles)	137,649	16,572	87.9%

Table 1. Test program statistics

Fault coverage results for every component along with the total processor fault coverage are illustrated in Table 2. The tick marks denote the targeted components by the proposed methodology and the methodology of [4].

	Fault coverage (%)			
Component	***Method of [4]***		***Proposed method***	
ALU	✓	98.48	✓	98.48
SHU	✓	94.08	✓	93.82
PC	✓	89.16		88.10
AC		99.33		98.67
IR		98.61		98.26
MAR		97.22		97.22
SR		98.88		92.13
CTRL		88.26		85.52
Total CPU		**91.42**		**91.10**

Table 2. Fault coverage results

The proposed methodology achieves a significant amount of reduction on key test program statistics like the number of processor instructions, the program size (the number of bytes to be downloaded by an external ATE to memory or stored in ROM) and the response data size (the number of test response bytes to be stored in memory and later to be unloaded by an external ATE or compressed by a test response analysis program) while it achieves virtually the same fault coverage results. It should be noted that conventional testing techniques (non software based) fault coverage results are presented in [4]. Full Scan and Logic BIST achieve total fault coverage 89.39% and 88.69%, respectively, while both techniques require circuit modifications. Also, they both result to area overhead (Logic BIST in particular) while Full Scan requires a high performance tester to apply at-speed.

It is evident that the proposed methodology is superior when compared to [4] with respect to the total memory requirements (test program size, data memory for test response storage). Furthermore, if one considers the additional memory requirements as a result of the self-test signature expansion into a large number of test vectors that have to be stored in memory [4], our proposed methodology memory requirements advantage

is further validated. (In [4], self-test signatures are expanded by the test generation program - software LFSR - into test vectors stored in memory, which subsequently are applied by the test application program).

The program (test code) size along with the response data size (the later applies when an on-chip test response analysis program is not employed to compress test responses into signatures) determines the total number of bytes that have to be downloaded and unloaded respectively, by the external ATE (for manufacturing testing) and thus relates directly to the tester time. Even if an on-chip test response analysis program is employed, its execution time relates directly to the test response data size and thus affecting the tester time.

The superiority of the proposed methodology on the test program execution time is obvious. The proposed test program, by addressing carefully the testability of key functional modules (ALU, SHU) achieves an almost one order of magnitude reduction of the test application time when compared to [4].

With a minimal amount of information extracted from the processor instruction set and RT level description, our methodology compares favorably to the one of [4] with respect to engineering effort. The methodology of [4], considering every component fault at the gate level, requires much larger engineering effort to perform repetitive, time consuming, LFSR pseudorandom constrained test generation and fault simulation in an iterative method at the test preparation stage. Furthermore, in [4], in case of using ATPG based stored test patterns for some components, constrained based ATPG at the gate level must be performed, where instruction set imposed constraints must be extracted, requiring a large engineering effort as well.

4. Conclusions

We proposed a software based self-test methodology for embedded processor cores that enables at-speed testing and achieves high fault coverage with no hardware overhead and performance degradation. The methodology targets processor components and applies deterministic data patterns (operands) for every component operation. When compared with existing software based self test methodologies (pseudorandom and ATPG based) it requires much less computational effort while it achieves a guaranteed high fault coverage facing the most hard to test functional modules favorably. We have demonstrated its effectiveness in the same processor core used in a recently published work [4]. The superiority of the proposed methodology in terms of both test program size, memory requirements and test application time is significant.

References

[1] International Technology Roadmap for Semiconductors, 1999 Edition

[2] T.G.Foote, D.E.Hoffman, W.V.Huott, T.J. Koprowski, B.J. Robbins and M.P. Kusko, "*Testing the 400 MHZ IBM Generation-4 CMOS Chip*", in Proceedings of the International Test Conference 1997, Washington DC., pp.106-114

[3] G.Hetherington, T.Fryars, N.Tamarapalli, M.Kassab, A.Hassan and J.Rajski, "*Logic BIST for large industrial designs: Real issues and case studies*", in Proceedings of the International Test Conference 1999, Atlantic City, NJ, pp.358-367

[4] Li Chen, S.Dey, "*Software-Based Self-Testing Methodology for Processor Cores*", IEEE Transactions on CAD of Integrated Circuits and Systems, vo.20, no.3, pp. 369-380, March 2001.

[5] J.Shen, J.Abraham, *"Native mode functional test generation for processors with applications to self-test and design validation"*, in Proceedings of the International Test Conference 1998, pp. 990-999.

[6] K.Batcher, C.Papachristou, *"Instruction randomization self test for processor cores",* in Proceedings of the VLSI Test Symposium 1999, pp. 34 – 40.

[7] J.Rajski, J.Tyszer, *"Arithmetic Built-In Self-Test for Embedded Systems",* Prentice Hall, 1997.

[8] F.Corno, M.Sonza Reorda, G.Squillero, M.Violante, *"On the Test of Microprocessor IP Cores"*, Design Automation & Test in Europe 2001, Munich, Germany, March 2001.

[9] D.Gizopoulos, A.Paschalis, Y.Zorian, *"An Effective Built-In Self-Test Scheme for Array Multipliers"*, IEEE Trans. Computers, vol. 48, no. 9, pp. 936-950, September 1999.

[10] D.Gizopoulos, A.Paschalis, Y.Zorian, *"An Effective Built-In Self-Test Scheme for Booth Multipliers"*, IEEE Design & Test of Computers, vol. 15, no. 3, pp. 105-111, July-September 1998.

[11] D.Gizopoulos, A.Paschalis, M.Psarakis, Y.Zorian, *"*An *Effective BIST Scheme for Arithmetic Logic Units"*, in Proceedings of the International Test Conference, 1997, Washington DC, pp. 868-877.

[12] D.Gizopoulos, A.Paschalis, Y.Zorian, *"An Effective BIST Scheme for Datapaths"*, International Test Conference 1996, pp. 76-85.

[13] A.Paschalis, D.Gizopoulos, N.Kranitis, M.Psarakis, Y.Zorian, *"Deterministic Software-Based Self-Testing of Embedded Processor Cores"*, Design Automation & Test in Europe 2001, Munich, Germany, March 2001.