Low-Cost Software-Based Self-Testing of RISC Processor Cores

N. Kranitis [1] G. Xenoulis [2] D. Gizopoulos [2] A. Paschalis [1] Y. Zorian [3]

[1] *Department of Informatics & Telecom. University of Athens, Greece* – {nkran|paschali}@di.uoa.gr
[2] *Department of Informatics, University of Piraeus, Greece* – {gxen|dgizop}@unipi.gr
[3] *Virage Logic, Fremont, CA, USA* – zorian@viragelogic.com

Abstract

Software self-testing of embedded processor cores which effectively partitions the testing effort between low-speed external equipment and internal processor resources, has been recently proposed as an alternative to classical hardware built-in self-test techniques over which it provides significant advantages.

In this paper we present a low-cost software-based self-testing methodology for processor cores with the aim of producing compact test code sequences developed with a limited engineering effort and achieving a high fault coverage for the processor core. The objective of small test code sequences is directly related to the utilization of low-speed external testers since test time is primarily determined by the time required to download the test code to the processor memory at the tester's low frequency.

Successful application of the methodology to a RISC processor core architecture with a 3-stage pipeline is demonstrated.

1 Introduction

The widely accepted System-on-Chip (SoC) design paradigm consists in most cases of one or more embedded processor cores surrounded by other cores for program and data storage, communication with peripherals, analog and mixed-signal functions, etc. Although SoC designs provide a significant step towards enriching functionality and reducing development time, their testing appears to be a difficult task. Test data volume required for external testing of SoC designs is excessive while the increasing gap between tester frequencies and SoC operating frequencies makes at-speed testing almost infeasible.

The opposite extreme of external testing, hardware built-in self-test moves the testing task from external test equipment to internal resources, synthesized particularly for this purpose. At-speed testing is achieved, while overall test costs are reduced [1]. Unfortunately, hardware self-test for embedded processors adds significant area and performance penalties that can not be afforded in carefully performance and power-optimized processor designs.

As an efficient alternative, software-based self-testing methodologies for embedded processors have been proposed. Software-based self-testing has a non-intrusive nature since it utilizes processor functionality and instruction set to perform self-testing. Thus, it does not add performance, area or power overheads in an optimized processor design. Self-testing approaches have been proposed in the literature, [2]-[8] and a review of some of them is given in [9]. An outline of the embedded software-based self-testing concept is shown in Figure 1.

Software-based self-testing of processor cores partitions the test resources needed to test a processor between tester memory for test program/data storage, and internal processor resources, i.e. its instruction set and functional modules. Software-based self-testing can be proved to be a *low test cost* methodology for processors as we show in this paper, and if appropriately applied, it has both *low test development cost* and *low test application cost*. The objectives of an effective software-based self-testing methodology for processor cores are: (a) *high fault coverage*, (b) *small test program and data size*, (c) *small test execution time*. The last two objectives, determine the overall test application time and the dominating one is the size of the test program and data since they are both downloaded from the (potentially low speed) tester in the on-chip memory, while execution of the test program is performed at higher processor speed.

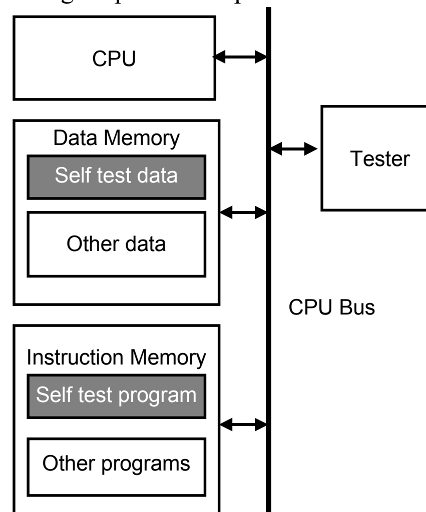

Figure 1: Software-based self-testing concept outline

1530-1591/03 $17.00 © 2003 IEEE

An important issue in software self-test program development is whether gate-level implementation details of the processor are availiable. A generic software-based self-testing methodology should only be based on high RT-level description of the processor and its instruction set architecture, thus providing a technology independent test development strategy. That applies both to the case where low level netlist is not available ('hard' IP versions) or in the case where technology re-mapping is required.

Previous approaches proposed in [2]-[5] apply *functional* self-testing to processor cores and rely on the use of pseudorandom instruction sequences and operations/operands. Due to the high level of abstraction of the approaches and their pseudorandom nature, structural fault coverage is usually low, although test programs with excessively large execution times are used.

In [6], a well-developed *structural* testing methodology for processor cores was presented. At the first stage, the *test preparation stage*, pseudorandom pattern sequences are developed for each processor component in an iterative method taking into consideration the constraints imposed by its instruction set. Subsequently, test sequences are encapsulated into self-test signatures that characterize each component and consist of the seed and the configuration of the pseudorandom TPG, along with the number of test patterns. Alternatively, component tests can be extracted by structural ATPG and downloaded directly in embedded memory by the tester. At the second stage, the *test application stage*, the component self-test signatures are first expanded on-chip by a software-emulated LFSR (test generation program) into pseudorandom test patterns, then stored in embedded memory and finally applied to the component by software test application programs. The pseudorandom nature of the approach of [6] leads to large self-test program, large memory requirements and excessive test application time. This methodology can be applied to relatively small processor cores [6] and only in the case that the gate-level netlist of the processor is available. It has been noted in [6] that architectural constraint extraction is a difficult task and that not all such constraints can be extracted from structural descriptions. In the case of larger "real" processors and even if a netlist is available, constrained test generation for the processor components using sequential ATPG is a very time consuming task, which may either lead to an unacceptable low fault coverage, or may generate a very large test set to achieve an acceptable one. Experimental results provided in [6] prove the superiority of software-based self-testing for processors over both Full Scan and hardware Logic BIST.

In [7], [8] we introduced a *high level structural* software-based self-test methodology for embedded processor cores, showing that small deterministic test sets, applied by compact test programs provide significant improvement when applied to the same simple processor design, Parwan, that was used in [6]. Compared to [6], the methodology described in [7], [8] requires about 20% smaller test program, 75% smaller test data and almost 90% smaller test application time (test cycles). Both methodologies achieve a single stuck-at fault coverage slightly higher than 91% for Parwan.

In this paper, we investigate the test quality that a low cost *greedy* software-based self-test methodology can achieve when applied to a complex processor core implementing a popular RISC architecture of significant size and complexity. Our methodology is high level because it is based on the Instruction Set Architecture (ISA) of the processor and its *Register Transfer (RT) level* description and does not require either architectural constraint extraction or sequential ATPG for the processor components. The RT level description is almost always available and is much more easily manageable than a detailed gate-level netlist. This is a very important parameter when "real" complex processors are considered.

The proposed methodology has two main objectives both aiming to achieve *low test cost*: (a) *the generation of as small as possible code sequences* with (b) *as small as possible engineering effort and test development time*. The first objective increases the efficiency of test resource partitioning by software-based self-testing: achieving an acceptable fault coverage with a small test sequence leads to smaller download times at the low frequency of the external tester. The second objective reduces test development cost and time-to-market.

The key points for the success of the proposed methodology are: (a) appropriate simple categorization of processor components in classes with the same properties and component prioritization for test development, b)development of self-test routines based on compact loops of instructions, (c) development of a "library" of small test sets that provide very high fault coverage for most types and architectures of the processor components. We remark that the approach proposed in [6], although it is based on structural testing targeting processor components, does not include any component classification part so that components are given different priorities for test program development.

Section 2 describes the proposed methodology, the component classification, the test priority criteria and the component test development. Section 3 describes the RISC processor case study. Experimental results presented in Section 4 demonstrate that the two low test cost objectives of the proposed methodology, small test development time and small test application time, are satisfied. Finally, section 5 concludes the paper.

To our knowledge this is the first paper that achieves complete application of a software-based self-testing methodology to a processor core of significant size and complexity and leads to a single stuck-at fault coverage of more than 92% without any performance or area penalty and with a self-test code size of approximately 1K words.

2 Proposed Software-Based Self-Test Methodology

Figure 2 illustrates the various steps of our low cost component-based self-test methodology which follows a top-down approach. We study the processor as an RT level design, consisting of a set of components. First, we classify the components in three classes: *functional*, *control* and *hidden* to evaluate their importance for testing of the entire processor and we target them in a descending order of importance so that the larger and easily accessible (therefore easily testable) components are targeted first and a high fault coverage is achieved with reasonable effort. The resulting self-test program is very small since small routines consisting of compact loops are used for each of the components.

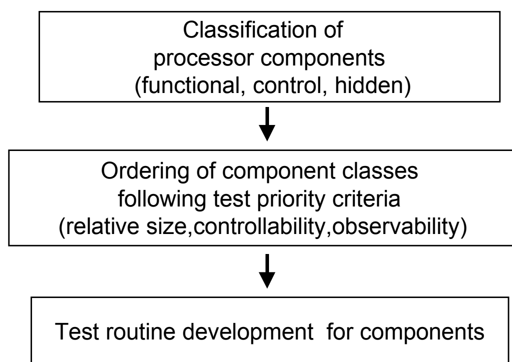

Figure 2: Proposed methodology outline

The proposed methodology does not require constraint extraction for the processor or sequential ATPG. It can be characterized as a *greedy* methodology seeking the highest possible fault coverage targeting the largest and more easily testable processor components first. For this reason, its application provides a low test cost solution, i.e. a testing strategy with small test development cost and small test program application cost. Application of the methodology to the target RISC processor of this paper was very successful. The limit of the applicability of the methodology may appear in processor cores where significant parts of the architecture (focusing performance improvement) are not easily accessible by processor instructions.

2.1 Processor components classification

We classify the components that appear in a processor core RT level description in the following three classes.

- *Functional components*. The components of a processor that are directly related to the execution of instructions and their existence is directly implied by the format of one or more instructions. Such components are usually the Arithmetic Logic Unit (ALU), the shifter, the multiplier, the register file, etc. These components either store data or perform a specific function on data.
- *Control components*. The components that control either the flow of instructions/data inside the processor core or from/to the external environment (memory, peripherals). Such components are usually the program counter logic, instruction and data memory control registers and logic, etc. These components are not directly related to specific functions of the processor and their existence is not implied by any instruction format of the processor.
- *Hidden components*. The components that are added in a processor architecture usually to increase its performance but they are not visible to the assembly language programmer. They include pipeline registers and control and other performance increasing components related to Instruction Level Parallelism (ILP) techniques, branch prediction techniques, etc.

2.2 Test priority criteria

For a low-test cost high-level software-based self-testing methodology that aims to develop small test programs in a small test development time the three classes of components have different test priorities. By test priority we mean the order in which test programs will be developed for each component. High priority components will be considered first while low priority components will be considered afterwards and *only if the achieved fault coverage is not sufficient*. The characteristics of a module that determine its priority in our methodology are its relative size and the controllability and observability of the component by processor instructions.

First of all, we deal with the processor components that have the *largest contribution to the overall processor fault coverage*. As it is obvious, these are the largest components of the processor, which include most of the stuck-at fault of the entire processor.

If exact sizes (gate counts) of the components of the processor are available then their relation is known and components of the same class (functional, control, hidden) are sorted in descending sizes.

If the processor core is available in a hard form, i.e. neither gate netlist nor a synthesizable HDL model of the processor is available, then this data could be obtained from the processor IP supplier. In the worst case that such data cannot be available in any way, then it is reasonable to make some assumptions on the component sizes. The following three observations are in most cases valid.

- The *register file* of the processor is one of the largest components. This fact is particularly true in RISC processors with a load/store architecture, which have a large number of general-purpose registers.
- The *hardware multiplier and/or divider*, if exist, are also two of the largest components. This fact is particularly true in RISC processors as well as in DSPs. We note that the methodology is very well applied to DSP cores.

- The *functional components* of the processor that perform all arithmetic and logic operations of the processor are much larger than the corresponding *control logic* which controls their operation. This size difference got larger when processor generations moved from internal busses of 8-bits and 16-bits to those of 32-bits or 64-bits. Additionally, in DSPs where many functional components of the same type co-exist, the processor size dominating factor is the size of the functional components.

Therefore, the first class of processor components, the functional components, have the highest test priority due to their relative size.

The other characteristic of a module that determines its priority in our methodology is the controllability and observability of the component by processor instructions, i.e. how easy it is to apply a specific test input to the component and how easy it is to observe its outputs at the processor primary outputs. The definitions of controllability and observability of a processor component are the following.

Processor component controllability: the shortest instruction sequence required to apply a test pattern to the component inputs.

Processor component observability: the shortest instruction sequence required to propagate a component outputs to the primary outputs of the processor.

Processor components may have two different types of inputs and outputs, *control inputs/outputs* and *data inputs/outputs*. Control inputs/outputs come from control components towards functional components or the other way round. Data inputs/outputs come from internal registers or from the input/output data ports of the processor core. Component controllability and observability related to data inputs/outputs is higher than to control inputs/outputs and hence functional components have higher controllability and observability than control and hidden components.

Table 1 summarizes the controllability and observability of the three classes of components defined earlier and the priority given to their test development by the proposed methodology.

Component Class	Controllability/ Observability	Test Priority
Functional	High	High
Control	Medium	Medium
Hidden	Low	Low

Table 1: Component classes test priority

Therefore, according to our methodology, the functional components of the processor have the highest test priority for test development and are targeted first since their size dominates the processor area and have high observability and controllability. The second and third step of proposed methodology outlined in Figure 2, are expanded and illustrated to Figure 3. Phase A develops test program for the functional components, Phase B for the control and Phase C for the hidden components.

As we will demonstrate in the next sections, the proposed methodology provides very high fault coverage for a complex processor model, even when test program development *is performed only for Phase A*. Phase A is the one that deals with the high priority components, i.e. functional components, the required engineering effort is very low while the achieved overall test cost is also very low as justified by the very small test program size and the very small test program execution time.

When moving from Phase A to Phase B and to Phase C component test development complexity increases since the target components are of decreasing accessibility with processor instructions. Therefore, if a high fault coverage (above an acceptable level) is gained only targeting Phase A components then the necessary test development effort is very low while the resulting test program is very small. As we see in the next sections this is the actual situation in the RISC processor core studied.

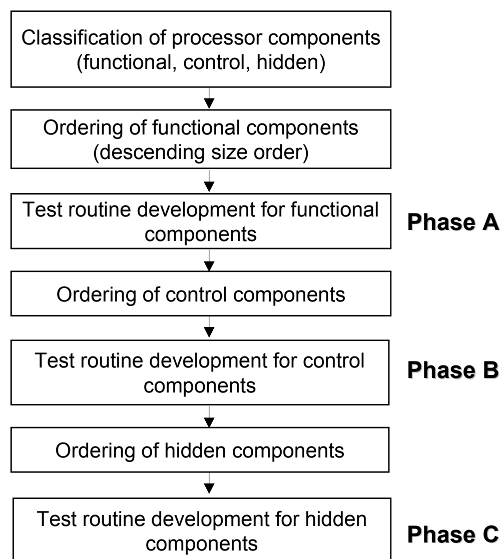

Figure 3: Proposed methodology expanded to phases

2.3 *Test routine development*

In this subsection we elaborate on the test routine development at the component level which is performed as shown in Figure 4. The proposed methodology is component-based and thus it deals separately with component operations as they are extracted from the RT level description of the processor. The first step of the component level test development shown in Figure 4 is the identification of the operations of the component under test. At the second step, the set of instructions, which excite the component operations is identified. The final step at the component level is to develop self-test routines based either on known or new test sets for the components.

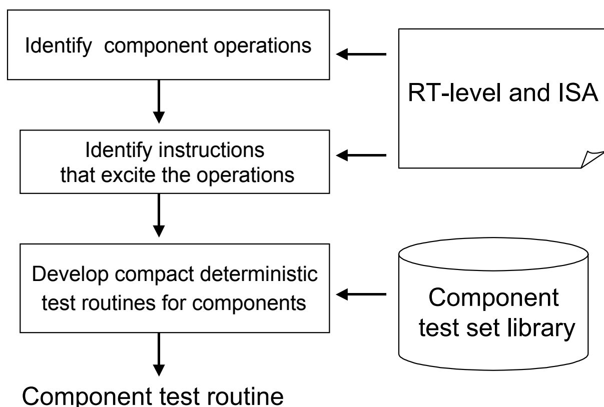

Figure 4: High-level component-based test development

Most processor components (particularly the functional components like registers, register files, arithmetic and logic units) have a very regular or semi-regular structure (arrays of identical cells, tree-like structures of multiplexers/demultiplexers, memory element arrays, etc) which can be efficiently tested with small and regular test sets. Major consideration in test development at the component level is that the derived programs must be small and have a small number of clock cycles for their execution. Both these factors lead to reduced test application time.

3 The Plasma/MIPS Processor Model

We applied the proposed methodology to a recently developed RISC processor model called *Plasma/MIPS* CPU core, which supports interrupts and all MIPS I user mode instructions except unaligned load and store operations (which are patented) and exceptions. The CPU core is implemented in VHDL with a 3-stage pipeline [10].

Table 2 shows the classification of the Plasma/MIPS components in the classes described earlier. Apart from the components shown in Table 2, few gates/flip-flops surround them. We call these: Glue Logic.

Component Name	Component Class
Register File	Functional
Multiplier/Divider	Functional
Arithmetic-Logic Unit	Functional
Barrel Shifter	Functional
Memory Control	Control
Program Counter Logic	Control
Control Logic	Control
Bus Multiplexer	Control
Pipeline	Hidden

Table 2: Plasma/MIPS components classification

4 Experimental results

The Plasma/MIPS processor VHDL model has been synthesized in a 0.35 um library and led to a design operating at 66 MHz frequency. The resulting gate counts for each component and the processor overall are shown in Table 3. A 2-input NAND gate is the gate count unit. Mentor Graphics tools were used for VHDL synthesis, functional and fault simulation (Leonardo, ModelSim and FlexTest products, respectively).

Component Name	Gate Count
Register File (RegF)	9,906
Multiplier/Divider(MulD)	3,044
Arithmetic-Logic Unit(ALU)	491
Barrel Shifter (BSH)	682
Memory Control (MCTRL)	1,112
Program Counter Logic (PCL)	444
Control Logic (CTRL)	223
Bus Multiplexer (BMUX)	453
Pipeline (PLN)	885
Glue Logic (GL)	219
Plasma/MIPS Processor	17,459

Table 3: Plasma/MIPS components gate counts

We have applied the proposed methodology to the Plasma/MIPS processor core and developed different self-test programs for the components starting with the four components of the first class, the functional components (Phase A). The self-test program statistics are given in Table 4 in number of 32-bit words and number of clock cycles for program execution.

	Phase A	**Phase A+B**
Test Program (words)	885	965
Clock Cycles	3,393	3,552

Table 4: Self-test programs statistics

Fault simulation results after the application of the Phase A developed test programs to the processor netlist are illustrated in columns 2 and 3 of Table 5. The components that have been targeted for Phase A test development are marked in Table 5. According to our methodology the highest priority components for test development are the four functional components of the Plasma/MIPS processor, for which four different test programs were developed. The MOFC column of Table 5 shows the percentage of the processor overall fault coverage which is missed at each of the components.

It is useful to note that the fault coverage results obtained for the very simple Parwan processor in [6], [7], [8] are a little higher than 91%, while fault coverages obtained by earlier software-based self-testing approaches were even lower.

Component Name		Phase A % FC	Phase A % MOFC		Phase A+B % FC	Phase A+B % MOFC
RegF	√	97.7	1.2	√	97.7	1.2
MulD	√	87.5	2.4	√	87.5	2.4
ALU	√	96.6	0.1	√	96.6	0.1
BSH	√	98.4	0.1	√	98.4	0.1
MCTRL		70.5	1.8	√	88.3	0.7
PCL		53.1	1.2		53.1	1.2
CTRL		77.5	0.4		78.9	0.4
BMUX		65.5	1.3		65.7	1.3
PLN		91.9	0.2		91.9	0.3
GL		94.3	0.1		96.9	0.1
Plasma		91.1	8.9		92.2	7.8

Table 5: Fault coverage on Plasma/MIPS with successive Phase test development

In order to show how multiple phase test development proceeds and the trade-off between increased fault coverage and additional test cost in test program size and execution time we consider Phase B, that is test development for control class components. Moving to Phase B component test development, the Memory Controller component is targeted first since it has high test priority among the control class components due to its largest size (see Table 3) and greatest MOFC after Phase A test application (see Table 5). Fault coverage results after Phase B test development are illustrated in the two rightmost columns of Table 5. If the fault coverage results are not satisfactory, the rest of the components that follow the Memory Controller in the control class ordering should be targeted next but at an increasingly test cost. In this case study, the only hidden class component, the pipeline component, is tested satisfactory.

A key point in the success of the proposed methodology is the efficiency of the library test sets both in terms of test program compactness and fault coverage. The Phase A results of Table 4 and Table 5 are based on the use of a few deterministic (not ATPG generated) test patterns which take advantage of the regularity of each processor functional component architecture. These patterns are applied by compact self-test routines with high structural test coverage. The small test program size has a direct impact on the download time from tester to the on-chip memory and usually dominates the total test time, while the other total test time component, test execution time (clock cycles) is also very small. The comparison with other software self-test approaches is favorable since they would require long pseurorandom pattern sequences application (increased test execution time) for random pattern resistant components, or large amount of test data (structural ATPG patterns) downloaded from tester to on-chip memory (long test download time).

We obtained very similar fault coverage results when the processor was synthesized in a different technology library. This fact demonstrates that the proposed methodology is efficient in targeting different gate-level netlists by exploiting RT-level architectural characteristics driving down the test development cost.

5 Conclusions

We have described a software-based self-testing methodology for processor cores and applied it to a complex RISC processor architecture. The proposed methodology does not require either manual or automatic constraint extraction for the processor components, or any sequential ATPG. It can be applied when just an RT level description and the instruction set architecture of the processor are known. The methodology aims to construct small and fast self-test programs to achieve an overall low test cost for the processor, i.e. low test development and low test application costs. Utilization of low-speed, low-cost external testers is highly improved by the proposed methodology, since download time from tester to chip done at the low frequency of the tester is very small. The successful application of the methodology on the complex architecture of the Plasma/MIPS model has been analyzed.

References

[1] ITRS, 2001 edition, http://public.itrs.net/Files/2001ITRS/Home.htm

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

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

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

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

[6] 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.

[7] N.Kranitis, A.Paschalis, D.Gizopoulos, Y.Zorian, *"Effective Software Self-Test Methodology for Processor Cores"*, in Proc. of the Design Automation & Test in Europe 2002, pp. 592-597.

[8] N.Kranitis, D.Gizopoulos, A.Paschalis, Y.Zorian, *"Instruction-Based Self-Testing of Processor Cores"*, in Proceedings of the IEEE VLSI Test Symposium 2002, pp. 223-228.

[9] A.Krstic, L.Chen, W.-C.Lai, K.-T.Cheng, S.Dey, *"Embedded Software-Based Self-Test for Programmable Core-Based Designs"*, IEEE Design & Test of Computers, July-August 2002, pp. 18-26.

[10] Plasma CPU Model. http://www.opencores.org/projects/mips