

# Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores

N. Kranitis<sup>1</sup> G. Xenoulis<sup>2</sup> A. Paschalis<sup>1</sup> D. Gizopoulos<sup>2</sup> Y. Zorian<sup>3</sup>

<sup>1</sup>Department of Informatics & Telecom, University of Athens, Greece – {nkran|paschali}@di.uoa.gr

<sup>2</sup>Department of Informatics, University of Piraeus, Greece – {gxen|dgizop}@unipi.gr

<sup>3</sup>Virage Logic, Fremont, CA, USA – zorian@viragelogic.com

## Abstract

Embedded processor testing techniques based on the execution of self-test routines, have been recently proposed as an effective alternative to classical hardware Built-In Self Test. Software-based self-testing provides at-speed testing capability and does not add hardware or performance penalties. It efficiently partitions the testing task between external testers and internal processor resources.

In this paper we analyze the application of a software-based self-testing methodology to different implementations of a complex embedded processor architecture. We demonstrate that such a methodology provides high test quality in different processor implementations with low test development and low test application costs.

## 1 Introduction

Consumer demands for high performance and rich functionality have driven semiconductor manufacturing to the need for integration of multiple complex components on a single chip. Deep submicron technologies along with the use of the System-on-Chip (SoC) design paradigm made this possible. System development based on the use of a core-based architecture, where cores are interconnected by an industry standard on-chip bus is very common. This design strategy has been proved to be effective in terms of development time and productivity since it re-uses existing intellectual property (IP). Unfortunately, this benefit does not come at zero expenses. Approaching the multi-million gate SoC, design and test engineers face signal integrity problems, serious power consumption concerns and maybe more than these serious testability challenges.

Manufacturing testing of an SoC architecture built around one or more processor cores is 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. Moreover, external tester's accuracy problems can lead to serious yield loss when external testing is applied.

Traditional hardware BIST moves the testing task from external resources to internal hardware, synthesized for this purpose. At-speed testing is achieved, while overall test costs are reduced [1]. Unfortunately, hardware self-test for processors adds significant area and performance penalties that can not be afforded in carefully performance and power-optimized processor designs. This is a serious problem, in particular for embedded applications where performance and/or power consumption are very important factors. Moreover, application of hardware self-test is further limited because in many cases embedded processors are delivered in their 'hard' versions where no design changes are allowed at all.

Different approaches that can be grouped together under the term, software-based self-testing (SBST) for embedded processors have been recently proposed as an effective alternative to hardware self-test. Software-based self-testing has a non-intrusive nature since it utilizes existing processor resources and instruction set to perform self-testing. Therefore, SBST can potentially provide sufficient testing quality without any impact on performance, area or power consumption. Software-based self-testing approaches have been proposed in the literature, [2]-[10] and a review of some of them was given in [11]. Comparisons provided in [8] prove the superiority of SBST for processors over both Full Scan and hardware Logic BIST. An outline of the embedded SBST concept is shown in Figure 1.

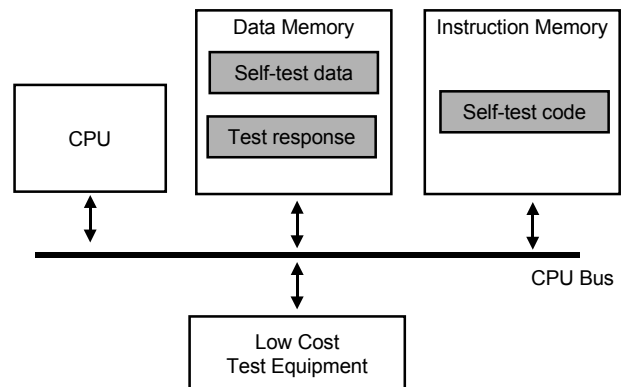


Figure 1: Software-based self-testing concept outline

Self-test routines and data are downloaded into instruction and data memories respectively, from a low-speed, low-cost external mechanism. Subsequently, these self-test routines are executed at-speed and test responses are stored back in on-chip RAM. These test results may be in a compacted or uncompact form and can be unloaded by the low cost external ATE for manufacturing testing. Execution from on-chip cache memory guarantees that no external bus cycles are initiated [10] and thus test application time is reduced. As an alternative, self-test routines may be executed from an on-chip ROM dedicated to this task when periodic on-line testing is performed for the device.

The SBST approaches presented so far in the literature can be classified in two different categories. The first category includes those approaches [2]-[5] that are functional in nature, sometimes without an a-priori fault model, where test development is performed at high level without any knowledge of the processor structure. Such approaches have the advantage of low test development cost but they also have the drawback that they provide low test coverage using large instruction sequences due to their high abstraction level and their pseudorandom philosophy as well. The derived test program has large size and requires excessive test application time. In addition, long fault simulation time is required for fault grading. The second category of approaches [6]-[8], are structural in nature, targeting processor components and fine-tuning the test development to the low, gate-level details of the processor core. Such approaches provide satisfactory test coverage results using pseudorandom and deterministic ATPG when the gate-level structure of the processor is available. The pseudorandom approach in [8] does not consider the regular structure of critical processor components and hence it leads to large self-test routines, large memory requirements and excessive test application time even when applied to a very small processor model [8]. Nevertheless, the gate level based test strategy with constraint extraction, as it is presented in [8], is a generalized approach targeting every processor component. However, in the case of large "real" processors, constrained test generation for the processor modules using a sequential ATPG is a very time consuming task, if at all achievable since usually only combinational module constrained test generation can be easily handled by commercial ATPG tools, and may either lead to an unacceptable low fault coverage, or generate a very large test set to achieve an acceptable one.

SBST methodologies based on gate-level ATPG and test tools working at the gate-level demand significant test development cost, due to the tremendous increase of the circuit size and complexity. Thus, it is highly desirable that a SBST methodology is based on a high RT level description of the processor and its instruction set architecture, thus providing a low cost and technology independent test development strategy.

In [9] we introduced a *high level structural* SBST methodology for embedded processor cores, showing that small deterministic test sets, applied by compact test routines provide significant improvement when applied to the same simple processor design, Parwan, that was used in [8]. Compared to [8], the methodology described in [9] requires 20% smaller test program, 75% smaller test data and almost 90% smaller test application time. Both methodologies achieve a single stuck-at fault coverage slightly higher than 91% for the simple Parwan processor.

In this paper, we analyze a complete application of our SBST methodology [9] in various implementations of a real RISC instruction set architecture. We demonstrate how a high RT-level, component-based, SBST methodology can effectively provide very high structural fault coverage for different implementations of a comprehensive processor architecture. The applied SBST methodology is based on the high RT-level description of the processor and its instruction set architecture and thus it provides a technology independent test development strategy aiming to *low test cost* both in terms of small test development and small test application.

The effectiveness of the proposed methodology lies beneath the architecture of most critical-to-test processor components, existing in all processor architectures. These components, if studied comprehensively, possess an inherent regularity within, which can lead to very efficient test algorithms for any gate-level implementation. This inherent regularity is absolutely not exploited neither by pseudorandom test development nor by ATPG-based test development approaches. The methodology seeks the highest possible fault coverage targeting the largest and most easily testable processor components first, with small and fast test routines. Thus, the methodology leads very quickly to a high structural fault coverage, with very low test development and test application cost.

Experimental comparison results targeting one of the most critical-to-test processor components, the parallel multiplier, demonstrate that the applied SBST methodology compares favorably with previous approaches when complex constraint extraction and constrained TPG (pseudorandom and structural ATPG) for such approaches is feasible.

Section 2 summarizes the proposed methodology, the processor components classification, the test priority criteria and test routine development. Section 3 describes the RISC processor model used in the experimental results. Section 4 demonstrates that the low test cost objective is satisfied as derived by the experimental results in different implementations of the instruction set architecture and provides comparison with previous SBST approaches. Finally, Section 5 concludes the paper.

## 2 Proposed High-Level Component-based SBST Overview

The key characteristics of the high RT-level component-based SBST methodology proposed in [9] and expanded in this paper are the following:

- A divide-and-conquer approach is applied using component-based test development.
- Test development is based only on the Instruction Set Architecture (ISA) of the processor and its RT-level description, which is in almost all cases available, without the need of low gate-level fine-tuning.

Although the main target always remains the high structural fault coverage (stuck-at), test cost should be considered as a very important aspect. The proposed methodology has two main objectives both aiming to *low test cost*: (a) *the generation of as small as possible code routines* with (b) *as small as possible engineering effort and test development time*. The first objective increases the efficiency of test resource partitioning by SBST since it leads to smaller download times at the low frequency of the external tester. The second objective reduces test development cost and time-to-market leading to significant improvements in product cost-effectiveness and market success.

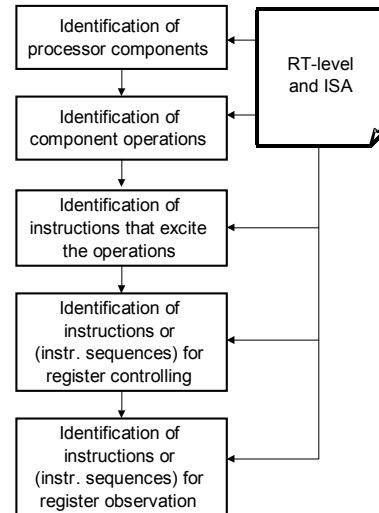
The high RT-level test development of the proposed approach is well suited to the high RT-level flow of the design cycle. Since design, simulation, and synthesis are usually carried out at the high RT-level, test development can also be carried out at the same level providing high convenience and flexibility. In this case processor cores can be easily integrated into an SoC environment, configured and re-targeted in a variety of silicon technologies without any specific need for fine-tuning the test development to specific synthesis optimization parameters using a specific technology library. Experimental results show that the gate-level independent proposed test strategy is very efficient and achieves more or less the same high fault coverage results for different gate-level implementations of the RTL processor core.

The application of the proposed approach to a processor core consists of the following three phases:

Phase A: Identification of processor components and component operations, as well as, instructions that excite component operations and instructions (or instruction sequences) for controlling or observing processor registers.

Phase B: Categorization of processor components in classes with the same properties and component prioritization for test development.

Phase C: Development of self-test routines with compact loops of instructions based on a “library” of small pre-computed test sets that provide very high fault coverage for most types and architectures of the processor components independently of word length.



**Figure 2.** Proposed SBST methodology (phase A)

These phases are explained in more details in the following subsections.

### 2.1 Phase A: Information extraction

The starting point of our component-based SBST test strategy is the instruction format (known from the processor ISA) and the processor micro-operations (known from the processor RTL description). Based on these two pieces of information the methodology:

- Identifies appropriate assembly language instruction sequences to control the values at the inputs of processor components under test (CUT); this is the controllability part of the methodology.
- Identifies appropriate assembly language instruction sequences to ensure propagation of the CUT outputs to the processor primary outputs; this is the observability part of the methodology.

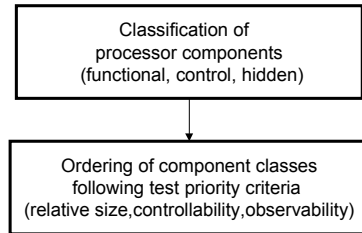
Both the control and observe processes can be performed using single processor instructions or an instruction sequence may be required.

From the processor instruction set architecture (ISA) and the RT-level description, we extract the following information, as summarized in Figure 2.

- (a) The set of all the processor components  $C$ .
- (b) The set  $O_C$  of all the operations of each component  $C$ , along with the corresponding control signals that the processor control unit drives to  $C$  for the execution of this operation.
- (c) The set of instructions  $I_{C,O}$  that, during their execution, enable the same control signals and drive component  $C$  to perform the same operation  $O$ .
- (d) Appropriate instructions or instruction sequences that control the content of processor registers
- (e) Appropriate instructions or instruction sequences that make the content of processor registers observable at primary outputs.

## 2.2 Phase B: Processor components classification and test priority

From the information extracting in the previous phase A, we classify and prioritize the processor components as summarized in Figure 3.



**Figure 3:** Proposed SBST methodology (phase B)

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

- *Functional components.* The components of a processor that are directly related to the execution of instructions and are in some sense visible to the assembly language programmer. Such components are usually the Arithmetic Logic Unit (ALU), the shifter, the multiplier, registers, the register file, etc. These components either perform specific arithmetic/logic operations on data or are storage elements for data. Our methodology acquires the information on the number and types of functional components from the RTL description of the processor.
- *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 evident in the 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.

For a low test cost, high-level SBST methodology that aims to develop small test routines in a small test development time the three classes of components have different test priorities. Test priority determines the order in which test routines 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 overall fault coverage result is not adequate. In most cases test development for top priority components leads, as a side effect, to sufficient fault coverage for not targeted components as

well. This is particularly true in processor architectures where the execution of instructions in functional units also excites many of the control components as well. The characteristics of a module that determine its priority in our methodology are the relative size and the controllability and observability of the component using 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 faults of the entire processor.

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 many general-purpose registers. Large register files offer many advantages enabling compilers to increase performance by reducing memory traffic.
- The *parallel multiplier*, if exists, is also one of the largest components. This fact is particularly true in RISC processors as well as in DSPs.
- 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.

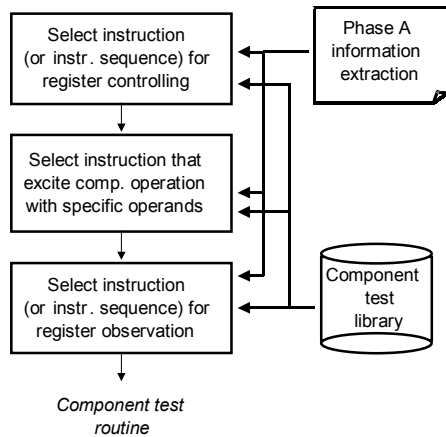
Therefore, the first class of processor components, the functional components, have the highest test priority due to their relative size. When fault coverage is not sufficient test development may proceed to the other two types of components. This can either be done using regular test sets if components have some kind of regularity, or, as an alternative, other test development approaches can be followed (structural, functional, etc).

The other characteristic of a module that determines its priority in our methodology is the accessibility (controllability and observability) of the component by processor instructions. The controllability of a processor component relates to how easy, an instruction sequence can apply a test pattern to the component inputs while the observability relates to how easy an instruction sequence propagates component output values to the primary outputs of the processor. Actually, the controllability (observability) of processor components is directly mapped to the controllability (observability) of the registers that drive (are driven by) the components inputs (outputs).

Therefore, according to our methodology, the functional components of the processor have the highest test priority for test development since their size dominates the processor area and they are easily accessible.

### 2.3 Phase C: 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.



**Figure 4:** Proposed SBST methodology (phase C)

The development of the dedicated *self-test routines* for testing specific processor components according to our test methodology [9] is performed in two steps as follows:

#### Step C1. Instruction selection.

In Phase A we have identified the set  $I_{C,O}$  which consists of processor instructions  $I$  that, during execution cause component  $C$  to perform operation  $O$ . The instructions that belong to the same set  $I_{C,O}$  have different controllability/observability properties since, when operation  $O$  is performed, the inputs of component  $C$  are driven by internal processor registers with different controllability characteristics while the outputs of component  $C$  are forwarded to internal processor registers with different observability characteristics. Therefore, for every component operation  $O_C$  derived in phase A, we select an instruction  $I$  from the set  $I_{C,O}$  that results in selecting the shortest instruction sequence required to apply the specific operand to component inputs and the shortest instruction sequence required to propagate the component outputs to the processor primary outputs. This way we avoid further constraint analysis.

#### Step C2. Operand selection.

In this step we consider the deterministic operands that must be applied to each component to achieve high structural fault coverage.

The key for selecting the most appropriate deterministic operands lies beneath the architecture of most critical-to-test processor components. Such components have an inherent regularity within, which can

lead to very efficient test algorithms for any gate-level implementation. This inherent regularity is not exploited either by pseudorandom test development or by ATPG-based test development approaches. Many processor components (in particular the functional components like registers, register files, arithmetic and logic operation modules, interconnect) have a very regular or semi-regular structure. Regular structure appears in the form of arrays of identical cells (linear or rectangular), tree-like structures of multiplexers, memory element arrays, etc. Such components can be efficiently tested with small and regular test sets that are gate-level independent, i.e. provide high fault coverage for any different gate-level implementation.

We have developed a component test library of precomputed small tests in the form of assembly pseudocode that provides very high fault coverage for most types and architectures of the processor components. Test routines consist of efficient compact loops, thus they require a very small number of bytes while the small number of tests results in low test routine execution time. The test routines in assembly pseudocode, are tailored to the processor's under test instruction set and assembly language. This is a part of the methodology that is already semi-automated and can significantly help the overall automation of the entire approach.

It is very important to note that constructing such a test library although could look like a very time consuming task, it is a one-time cost. Of course, the library can be enriched anytime with new tests i.e. that deal with new architectures implementing arithmetic/logic operations.

As already mentioned, it would be very desirable for a low cost test development in a SBST methodology to be gate-level independent. Since our methodology is component-based targeting structural (stuck-at) faults, it would be very desirable if implementation independent component tests were available. In fact, previous work endorses such an approach [12].

In what follows we describe how regular structures in functional datapath components that implement the most usual arithmetic, logic, interconnect, storage operations, can be efficiently tested with a few deterministic tests without the need of gate-level details.

#### Arithmetic components testing

For components that implement arithmetic operations, we have developed precomputed tests for every type of arithmetic operations like addition, subtraction, multiplication for different word lengths. Precomputed tests exist also for several architecture alternatives. For example, for the addition operation precomputed tests exist for ripple-carry adder (RCA), carry-look-ahead (CLA), etc architectures. Likewise, for the multiplication operation precomputed tests exist for carry-save array, booth-encoded, Wallace tree summation, etc architectures [13], [14].

### Interconnect / Multiplexer Testing

Interconnections between processor components and data flow in a processor datapath is guided by several multiplexers controlled by appropriate signals generated after instruction decoding. An n-to-1 multiplexer is usually decomposed and implemented by smaller multiplexers in a tree structure. For example an 8-to-1 mux can be classically implemented as a tree of seven 2-to-1 muxes. The implementation of a n-to-1 mux tree is not unique and in a RTL design, logic synthesis tools can generate different implementations depending on the technology mapping algorithms and technology libraries. For the 2-to-1 m-bit width mux, a test set provides complete single stuck-at fault testability, if it applies the following four input combinations: (S,A,B): (1,0...0,1...1), (0,1...1,0...0), (1,1...1, 0...0), (0,0...0, 1...1) where S is the select signal of the multiplexer and A, B are the two m-bit wide data inputs which are selected when S=0 and S=1, respectively. In general, for a n-to-1, m-bit wide mux, the deterministic test set is linear,  $2n$  test vectors are required and the single stuck-at fault coverage is 100%.

### Register File Testing

A register file in high RTL is constructed using HDL array declarations. The register file design can be considered as an hierarchical design with sub-components. Testing the two large multiplexer trees (usually 32 input, 32-bit wide muxes for processor architectures with 32 registers each 32-bit wide) that implement the two read ports along with the necessary tests for flip-flops and load enable muxes that implement the register storage elements results in near complete (>99.5 %) stuck-at fault coverage when synthesized in gates. We have developed test routines that apply the necessary tests and propagate the results to primary outputs. These routines are gate-level independent and have a size of only a few hundreds of bytes and test cycles.

### Logic Array Testing

Testing a logic array that implements multi-bit boolean logic operations like *and*, *or*, *nor*, *xor*, *not* is performed by applying the well known necessary patterns while propagating the test response to well observable registers and primary outputs. For example a multi-bit AND logic operation is tested by applying the following data patterns (A,B): (0...0, 1...1), (1...1,0...0), (1...1,1...1) where A, B are the multi-bit input busses.

The application of the proposed methodology is not restricted only to functional datapath components. In case that the fault coverage derived by targeting the functional datapath components is not adequate, component test development should continue with the other classes of components (i.e control, hidden). In this case control components are targeted as well. Control components that

consist of random logic may be very difficult to be tackled by small and fast routines and the test cost would grow resulting in a cost/coverage trade-off. However it is important to note that not-targeted components can be sufficiently tested as a side-effect of testing the targeted components. Furthermore, augmenting the compact test program with simple high level functional testing techniques like applying every instruction, RTL code coverage metrics, etc., results at even higher fault coverage results. The limit of the methodology applicability should be a specific processor architecture that contains large control or hidden components which are not sufficiently tested as mention above.

The key advantage of the proposed methodology is that it leads to a high fault coverage in very low test development (since it is based on high RT-level) and test application cost.

## 3 Case study: The Plasma/MIPS Architecture Processor Model

The proposed methodology is demonstrated through its complete application to different implementations of a public available RISC processor model, called *Plasma/MIPS* CPU core. It supports interrupts and all MIPS I™ user mode instructions except unaligned load and store operations (which are patented) and exceptions. The synthesizable CPU core is implemented in VHDL with a 3-stage pipeline [15].

We have enhanced the CPU model of [15] by adding a parallel multiplier to provide closer correspondence with real-world high-performance RISC processor core models. The fast multiplier module was generated using a public available module generator [16] and has the following characteristics: Booth recoding, Wallace trees for partial product summation and fast carry lookahead addition at the final stage. Table 1 shows the classification of the Plasma/MIPS components in the classes described earlier.

Component Name	Component Class
<b>Register File</b>	Functional
<b>Parallel Multiplier</b>	Functional
<b>Serial Divider</b>	Functional
<b>Arithmetic-Logic Unit (ALU)</b>	Functional
<b>Shifter</b>	Functional
<b>Memory Control</b>	Control
<b>Program Counter Logic (PC)</b>	Control
<b>Control Logic</b>	Control
<b>Bus Multiplexer</b>	Control
<b>Pipeline</b>	Hidden

Table 1: Plasma/MIPS components classification

## 4 Experimental results

The Plasma/MIPS processor VHDL model has been synthesized in two different technology libraries with different synthesis parameters. Synthesis A was optimized for area, targeted a 0.35  $\mu\text{m}$  technology library and the design runs at a clock frequency of 57 MHz. Synthesis B was optimized for area, targeted a 0.50  $\mu\text{m}$  technology library, and the design runs at a clock frequency of 42 MHz while Synthesis C optimized for delay, targeted a 0.35  $\mu\text{m}$  technology library, and the design runs at a clock frequency of 74 MHz.

Component Name	Syn. A 0.35 $\mu\text{m}$ 57 MHz	Syn. B 0.50 $\mu\text{m}$ 42 MHz	Syn. C 0.35 $\mu\text{m}$ 74 MHz
<b>Register File</b>	9,905	11,558	11,905
<b>Paral.Mult/ Serial Div</b>	11,601	11,654	13,358
<b>ALU</b>	491	558	900
<b>Shifter</b>	682	636	834
<b>Memory Control</b>	1,119	1,120	1,163
<b>PC Logic</b>	444	449	493
<b>Control Logic</b>	230	244	361
<b>Bus Multiplexer</b>	453	431	623
<b>Pipeline</b>	885	876	961
<b>Glue Logic</b>	270	298	283
<b>Plasma/MIPS Processor</b>	26,080	27,824	30,896

**Table 2:** Plasma/MIPS components gate counts

The resulting gate counts for each component and the processor overall, are shown in Table 2. A 2-input NAND gate is the gate count unit. Mentor Graphics suite was used for VHDL synthesis, functional and fault simulation (Leonardo, ModelSim and FlexTest products, respectively).

The components of highest priority for test development are the five functional components of the Plasma/MIPS processor. We developed different self-test routines for these functional components, since they represent about the 87% of the total processor area according to the proposed methodology. Although very high fault coverage (>94.5%) was achieved in a first phase, test program was enhanced by routines targeting the memory control and control logic components. This is a classic engineering approach since in any processor testing strategy the different addressing modes (realized by memory control component) as well as all the operation codes of the ISA (realized by the control logic module) should be excited and tested.

As an example we highlight test routine development for the ALU functional component. First, we identify the list of operations and corresponding control signals that processor components perform including the ALU case

study component. The ALU component has two 32-bit inputs, one 32-bit output and a control input that specifies the component operations that uses a 4-bit encoding. We identify the set of operations  $O_{ALU}$  that component ALU performs.  $O_{ALU} = \{\text{add, subtract, and, or, nor, xor, set\_on\_less\_than, set\_on\_less\_than\_signed}\}$ . Our target is to apply every operation  $o \in O_{ALU}$  using an appropriate instruction  $I \in I_{ALU,O}$ . According to the proposed test development, one instruction from each set is needed to apply the deterministic data operands that each operation imposes. For some of the 8 sets (i.e.  $I_{ALU,NOR}$ ) the mapping of the operation to a processor instruction is straightforward since there is only one processor instruction for every operation. Other sets (i.e.  $I_{ALU,OR}$ ,  $I_{ALU,XOR}$ ,  $I_{ALU,AND}$ ,  $I_{ALU,SUB}$ ,  $I_{ALU,SET\_ON\_LESS\_THAN}$ ,  $I_{ALU,SET\_ON\_LESS\_THAN\_SIGNED}$ ) consist of two instructions, one in R-type (register) and the other in the I-type (immediate) format. The  $I_{ALU,ADD}$  set consists of a large number of instructions since the ALU is used in memory reference instructions. For the last two kind of sets, instructions in the R-type format are used since they provide the best controllability and observability characteristics due to the use of the fully controllable and observable GPRs of the register file.

The self-test routine statistics are given in Table 3 in number of 32-bit words and number of clock cycles for routine execution. Along with total self-test routine statistics, partial statistics are given for each targeted component as well. The test cost in terms of test routine size and test application time is very small while test development is performed at high RT-level.

Component	Size (words)	%	Clock Cycles	%
<b>Reg. File</b>	319	37.4	582	10.0
<b>Paral. Mult.</b>	28	3.3	3122	53.9
<b>Serial Div.</b>	41	4.8	1154	19.9
<b>ALU</b>	79	9.3	275	4.7
<b>Shifter</b>	210	24.6	340	5.9
<b>Mem. Control</b>	76	8.9	160	2.8
<b>Contr. Logic</b>	100	11.7	164	2.8
<b>Total</b>	853	100.0	5,797	100.0

**Table 3:** Self-test routines statistics

Fault simulation results after the application of the developed test routines to processor netlists led to the fault coverage figures given in Table 4. The components that have been targeted for test development are marked on their left side in the table. The third, fifth and seventh columns of Table 4 show for each synthesis result (Synthesis A, B and C) the percentage of the processor overall fault coverage which is missing from each of the components.

Component Name	Synthesis A		Synthesis B		Synthesis C	
	% FC	% Missing overall FC	% FC	% Missing overall FC	% FC	% Missing overall FC
√ Register File	97.8	0.7	97.8	0.8	97.8	0.7
√ Paral. Mult/Serial Div	96.3	1.8	96.1	1.8	95.2	2.3
√ Arithmetic-Logic Unit	96.8	0.1	97.5	0.1	95.8	0.1
√ Shifter	98.4	0.1	99.9	0.0	95.3	0.2
√ Memory Control	87.9	0.5	88.5	0.4	90.3	0.3
√ Control Logic	89.3	0.1	88.3	0.2	85.3	0.3
Program Counter Logic	54.9	0.7	54.9	0.7	55.9	0.7
Bus Multiplexer	71.8	0.7	72.0	0.6	71.3	0.8
Pipeline	98.4	0.0	96.9	0.1	96.0	0.1
Glue Logic	97.8	0.0	97.4	0.0	94.9	0.1
Plasma/MIPS Processor	95.3	4.7	95.3	4.7	94.5	5.5

**Table 4:** Fault coverage on Plasma/MIPS with test development for targeted components

Downloading from ATE to memory, a test program of *less than 1K words*, one can reach at high fault coverage of *about 95%* very fast (fulfilling tight time-to-market) with very low test cost. Test development cost is minimized to the one-time cost of precomputed tests for components described at high RT-level, thus gate-level independent. Test application cost is proved very small since the small test program size is related directly to the download time from tester to the on-chip memory and usually dominates the total test time, while the other component of the total test time, test execution time (clock cycles) is also very small.

The fault coverage for the register file (97.8%) is less than the 99.5% that our test library provides for a general register file due to extra logic implemented including external interrupt handling, situation that software self-test can not handle without test instruction insertion [2]. The fault coverage of the parallel Booth recoded Wallace Tree multiplier is 99%. However, the fault coverage of the Parallel Multiplier/Serial divider component is 96% due to existence of control random logic within this complex module.

As it is shown in Table 4, we obtained very similar fault coverage results when the processor was synthesized in different technology libraries with different optimization scripts, optimizing for area or performance. This set of experiments was performed to show that test development for the targeted components which is performed at high RT-level results in high structural fault coverage that does not depend in the gate-level synthesis results. These experiments ensure that in the real design cycle where processor cores in a SoC environment, must be configured and re-targeted in different silicon technologies using different synthesis optimization parameters, test development is not fine-tuned each time

to the gate-level. Otherwise, the conventional gate-level specific ATPG method has to repeat the high cost and time-consuming test generation for every different implementation. This fact demonstrates that the proposed methodology is efficient in targeting different gate-level netlists where technology re-mapping is required by exploiting RT-level architectural characteristics driving down the test development cost. This is absolutely not the case in previously proposed approaches.

It is useful to note that the fault coverage results obtained for the simple Parwan processor (of 1.3K gates) in [8] and [9] are a little higher than 91%, while fault coverage results obtained by earlier software-based self-testing approaches [2]-[5] were even lower in approximately same sized small processor cores like the Intel 8051 microcontroller [5].

#### *Compaction of test responses*

The proposed methodology provides at-speed testing using low-cost external testers. Self-test routines and data are downloaded by a low cost external tester to the on-chip memory and the processor core executes the test program at-speed. The test responses which are stored back in on-chip memory are unloaded by the low-cost tester. The amount of test code downloaded from the tester to the on-chip memory and the number of test responses unloaded from the on-chip memory to the tester, determines the test time required to download/unload the test program and test responses respectively. This process dominates the total application time over the test execution time since the processor executes the test program at-speed. Thus, if the number of test responses is large, test response compaction may be required so that the external tester unloads only a very small number of compressed signatures.

The proposed methodology achieves very high fault coverage of *about 95%* by downloading a small test program of *less than 1K words (853 words)*. The number of test responses stored to on-chip memory after the execution of the self-test program is 610 words. This number is very small compared to other pseudorandom based SBST methodologies that would require application of a large number of pseudorandom patterns and storage of the test responses. Our test library provides a MISR test response compaction routine, that can be used to compress the targeted components test responses to signatures with negligible (less than 0.2%) aliasing probability, at the cost of downloading additional 58 words in our case study. Thus, when the MISR routine is used to compress the test responses stored in on-chip memory, the number of words downloaded is 911 words, the number of signatures unloaded is 7 words while the total test program test execution time becomes 14,427 cycles.

Since the test execution time depends on the processor speed and the processor cycles required to access the on-chip memory, the use of test response compaction is a tradeoff, with parameters determined by the specific processor and the test strategy adopted. The proposed test strategy applies a small number of deterministic (not ATPG) based tests, thus test response compaction might not be necessary. However, this is not the case for pseudorandom based test strategies where the very large number of test responses normally would require compaction of test responses.

### Comparison with ATPG and pseudorandom SBST

The results of Table 3 and Table 4 are based on the use of a few deterministic (not ATPG) test patterns (see Section 2) which take advantage of the inherent regularity of each processor functional component architecture. These patterns are applied by compact self-test routines with high structural test coverage for any gate-level implementation. This inherent regularity is not exploited either by pseudorandom test development or by ATPG-based test development approaches.

The comparison with other software self-test approaches is favorable since even if complex constraint extraction and constraint TPG (pseudorandom and structural ATPG) was feasible, they would require application of long pseudorandom pattern sequences (increased test execution time, very long fault simulation time for fault grading) 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).

To validate the former argument, a set of experiments was performed targeting one of the most critical-to-test processor components, the parallel multiplier. Besides the significance of the parallel multiplier in the total processor area and fault coverage, the parallel multiplier was selected as a case study for another reason. Test development according to alternative gate level structural, software self-test approaches for the parallel multiplier was applicable in a feasible way. That is:

- ❑ Manual constraint extraction was performed so that pseudorandom component tests apply on independent inputs.
- ❑ For structural ATPG generated tests, a commercial ATPG tool that was able to generate test patterns under constraints, was provided with a set of manually extracted *constraints*

We remark that for other functional units, particularly the sequential ones like the Register file and the Serial Divider, manual constrained extraction and sequential ATPG is not an easy task, if possible at all.

A set of 168 test patterns was generated by constrained structural ATPG using a commercial tool for the multiplier. Two different test routines (TRs) were developed to apply the deterministic ATPG test patterns using different approaches.

- a) TR<sub>ATPG\_LOOP</sub> applies the ATPG test patterns to the parallel multiplier using a load-apply-store algorithm
- b) TR<sub>ATPG\_IMM</sub> applies the ATPG test patterns using the immediate addressing format available in all instruction set architectures

The first approach would lead to less TR code, but increased test application time, while the second one would lead to increased TR code, but fewer cycles for testing.

Apart from the ATPG generated structural patterns, a software LFSR was coded in assembly language to apply pseudorandom patterns to the parallel multiplier (similarly to the software LFSR used in [8]). Test routine statistics and fault coverage results for each TR along with the proposed methodology TR, TR<sub>PROPOSED</sub> are given in the following Table.

Test routine	Size (words)	Clock cycles	Fault coverage (%)
TR <sub>ATPG_LOOP</sub>	373	3,895	99.5
TR <sub>ATPG_IMM</sub>	1,521	2,546	99.5
TR <sub>LFSR</sub>	53	11,130	97.6
<b>TR<sub>PROPOSED</sub></b>	<b>28</b>	<b>3,122</b>	<b>99.0</b>

**Table 5:** Multiplier test routine statistics comparing with ATPG and Pseudorandom SBST

The conclusion behind the contents of Table 5 is, that even when:

- gate-level tuned, costly test development is performed,
  - constrained extraction and constrained based structural ATPG and pseudorandom test development is possible
- the test routine size and test application time of such low level structural approaches is severely increased.

If we consider the test routine size as the most important factor due to the tester download time, the proposed method provides the smallest test routine. Only the coding of the software LFSR provides similar test routine size, but at the expense of increased test application time. Furthermore, the pseudorandom approach fails to reach an acceptable fault coverage after application of 600 test patterns. The only solution would be multiple LFSR sequence application using different seeds increasing furthermore the test application time and test routine size. On the other side, test routines based on structural ATPG deterministic patterns can be considered an efficient solution only when the number of test patterns is very small.

## 5 Conclusions

We have analyzed and demonstrated the application of a high-level component-based software-based self-test methodology to a complex embedded processor architecture, the Plasma/MIPS model. The proposed SBST methodology does not require either manual or automatic constraint extraction for the processor components, or any sequential ATPG at the gate level. It can be applied when just an RT-level description and the instruction set architecture of the processor are available. The methodology aims to construct small and fast self-test routines to achieve an overall low test cost for the processor, i.e. low test development and low test application costs. Low-speed, low-cost external testers are excellent utilized by the proposed methodology, since the downloading from the tester memory to the on-chip memory, performed at the low frequency of the tester, is completed in very short intervals.

We have demonstrated that high test quality (more than 95% fault coverage) is achieved with low test development and test application costs for various implementations of the benchmark CPU used in our experiments.

## References

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

[2] J.Shen, J.Abraham, “*Synthesis of Native Mode Self-Test Programs*”, in *Journal of Electronic Testing: Theory and Applications (JETTA)*, Oct. 1998, pp. 137-148

[3] J.Shen, J.Abraham, “*Native Mode Functional Test Generation for Microprocessors with Applications to Self-Test and Design Validation*”, in *Proc. of the International Test Conference 1998*, pp. 990-999.

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

[5] 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, pp.209-213

[6] V.Vedula, J.Abraham, J.Bhadra, “*Program Slicing for Hierarchical Test Generation*”, in *Proceedings of the IEEE VLSI Test Symposium 2002*, pp. 237-243.

[7] J. Lee, J.H. Patel, “*Hierarchical Test Generation Under Architectural Level Functional Constraints*”, in *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, Sept. 1996, pp. 1144-1151

[8] 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.

[9] 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.

[10] P.Parvathala, K.Maneparambil, W.Lindsay, “*FRITS – A Microprocessor Functional BIST Method*”, in *Proceedings of the IEEE International Test Conference 2002*, pp. 590-598.

[11] 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.

[12] H. Kim, J. Hayes, “*Realization-Independent ATPG for Designs with Unimplemented Blocks*”, *IEEE Transactions on CAD of Integrated Circuits and Systems*, vo.20, no.2, pp. 290-306, February 2001.

[13] A. Paschalis, D.Gizopoulos, N. Kranitis, M. Psarakis, Y. Zorian, “*An Effective BIST Architecture for Fast Multiplier Cores*”, in *Proc. of the Design Automation & Test in Europe 1999*, pp.117-121

[14] D. Gizopoulos, A. Paschalis, Y. Zorian, “*An Effective Built-In Self-Test Scheme for Parallel Multipliers*”, in *IEEE Transactions on Computers*, vol. 48, no.9, pp. 936-950, Sept. 99

[15] Plasma CPU Model. <http://www.opencores.org/projects/mips>

[16] J. Pihl, E. Sand, Arithmetic Module Generator, <http://www.fysel.ntnu.no/modgen>