SOFTWARE-BASED TEST FOR NON-PROGRAMMABLE CORES IN BUS-BASED SYSTEM-ON-CHIP ARCHITECTURES

Alexandre M. Amory[1], Leandro A. Oliveira[1] and Fernando G. Moraes[2]
[1]Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFRGS) - av. bento gonçalves, 9500 - prédio 43412/bloco IV - Porto Alegre - Brazil - CEP 91501-970; [2]Faculdade de Informática - Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) - av. ipiranga, 6681 - prédio 30/bloco 4 - Porto Alegre - Brazil - CEP 90619-900 - moraes@inf.pucrs.br

Abstract: With the advance in hardware integration, system-on-a-chip (SoC) test activities using only automatic test equipments (ATEs) result in an expensive option. Hardware-based test may reduce the ATE dependency. However, hardware-based test imposes some constraints like area overhead and processing speed degradation. The main objective of this work is to investigate and evaluate a less intrusive test approach called software-based test. Software-based test uses an embedded processor as source and sink of the test, sending the test patterns and reading the responses. A new integrated design and test environment has been developed to automatically synthesize test programs to test non-programmable cores of SoCs. Some benchmarks ISCAS85 and ISCAS89 are used to evaluate the proposed methodology.

Key words: SoC test; Software-based test; computer aided test (CAT); LFSR reseeding.

1. INTRODUCTION

In recent SoC based systems the amount of test data transferred between automatic test equipments (ATEs) and devices under test is becoming too large. Even expensive state-of-the-art ATEs restrict the SoC test as a result of limited memory resources, narrow channel bandwidth and low speed.

Please use the following format when citing this chapter:
Amory, Alexandre, M., Oliveira, Leandro, A., Moraes, Fernando, G., 2006, in IFIP International Federation for Information Processing, Volume 200, VLSI-SOC: From Systems to Chips, eds. Glesner, M., Reis, R., Indrusiak, L., Mooney, V., Eveking, H., (Boston: Springer), pp. 165-179.

One known approach to overcome ATE limitations is to use hardware-based test (i.e. built-in self-test BIST) to generate patterns and to analyze the results at -speed. This approach reduces the ATE constraints and the test cost. However, BIST has some drawbacks[1]: (i) some circuits are resistant to random patterns, resulting into a low fault-coverage; (ii) since new modules are inserted into the system, the total area, operation frequency and power consumption are negatively affected.

Software-based test is an alternative approach to BIST and ATE. SoC devices usually contain, at least, one embedded processor and use bus-based interconnection to integrate several IP cores. We propose a test methodology to test non-programmable IP cores using an embedded processor. Since there are no new test modules added to the system, figures as area usage, speed and power are not changed. A possible drawback of the software-based test, when compared to the hardware-based test is the longer time needed to apply the patterns and/or analyze the results[1].

This work has two main objectives. The first one is the evaluation of software-based against hardware-based test. The second objective is to present the developed Computer Aided Test (CAT) tool to synthesize the test program and to integrate IP cores in the SoC.

Unlike previous approaches[2,3,4], this paper presents a new tightly integrated design and test methodology, including commercial and in-house tools. Moreover, the evaluation compares software-based test with hardware-based test, unlike[3] which compares software-based test with boundary scan.

This paper is organized as follows. Section 2 presents the state-of-the-art in software-based test. Section 3 presents the bus-based SoC architecture. Section 4 details the developed CAT environment. Section 5 focuses on the software-based test evaluation based on some ISCAS85/89 benchmarks. Section 0 concludes this paper and presents some directions for future work.

2. SOFTWARE-BASED TEST

Software-based test can minimize some of the BIST drawbacks discussed before. The following is a list of advantages of this approach:

- Ease to reuse and to modify the test strategy as it is implemented in software;
- No specific test controller is required, since a generic embedded processor is responsible for the test control and execution;
- Reduced (or none) area overhead due to the test patterns generation and response compaction implemented in software;
- Reduced (or none) speed/power degradation, as there are no additional test modules;

- Can be applied to test processors, memories, general cores and interconnection (bus);
- Reduced design time compared to BIST even considering automated tools since there is an additional manual process required to make the target core become BIST-ready[1];
- Test occurs in normal operational mode, eliminating the extra power consumption of BIST[5];
- Can apply and analyze at-speed test signals[5] to detect delay faults, alleviating the need of high-speed tester. Moreover, since the test is applied in normal operational mode, the system is not over-tested.

Possible drawbacks of the software-based test are:

- The SoC must have an embedded processor;
- Additional time to create the test program;
- Extra memory needed to store the test program and the deterministic test patterns;
- Increased test time when comparing to hardware-based test. BIST usually generates patterns/ compact responses in just one clock cycle. However, in software-based test patterns are provided by the processor to the cores, taking longer to execute the same task than BIST modules. On the other hand, software-based test may be faster than ATE based test due to the limited bandwidth[3].

Thus, the goal of this work is to automate the test program generation, reducing the project time and to evaluate quantitatively the last two drawbacks (i.e. the increased test time and memory requirements).

2.1 Processor Test

The first works on software-based test for processors were conducted in the 70s by Thatte and Abraham[6]. Software-based self-test of processors is a challenging issue due to: (i) the number of different hardware modules to be tested; (ii) the limited number of pins to access the processor; (iii) the existence of modules that can not be accessed directly, as interrupt controller, flag, exception handler; (iv) the huge variation of implementations and architectures. Recent works on software-based processor test can be found in [1,7].

2.2 Memory Test

Memory test is very simple to be implemented by a processor due to its regular structure. Well-know algorithms[8] using deterministic patterns, e.g. 55h and AAh, are used. Zorian and Ivanov[9] consider a ROM as a combinational circuit, where the address is the input and the memory content

is the output. An exhaustive test is performed reading all memory contents and compacting the output.

2.3 Non-Programmable Core Test

The embedded processor can execute a specific program to test each core of the SoC, which is the goal of this paper. This program has the following functions: (i) generate the test patterns; (ii) send these patterns to the cores through the communication path (e.g. bus or TAM); (iii) read the test responses; (iv) compact these responses. Since it is a software-based test, it is possible to use different algorithms to test each core.

The test program can emulate the behavior of a LFSR and MISR in software, generating pseudo-random patterns and compacting responses[10]. A disadvantage of emulating a LFSR or a MISR in software is the increasing test time, since it may need several clock cycles to produce one pattern. Other approaches using adders, subtractors and multipliers to generate patterns and/or to compact responses can be used. In [11] it is shown that these modules can be used to substitute LFSR, generating comparable fault coverage, with the advantage of generating a pattern for each instruction (e.g. add).

Huang *et al.* [2] developed a bus-based architecture with MIPS processor, PCI bus and VCI interface. Using this architecture, they evaluated the test time and fault coverage of some ISCAS89 benchmarks. Lai and Cheng[4] used the same architecture presented in [2] to evaluate test programs generated for four ISCAS89 benchmarks, using the DLX processor. The test program length is from 40 to 27000 bytes. The test time is from 94 to 30430 clock cycles. The results point to important test memory requirements and test time compared to hardware-based test.

Hwang and Abraham[3] developed a bus-based architecture with an ARM processor and Wishbone interface. The authors compared the test time and area overhead between software-based test and boundary scan. In both cases, the software-based test presented better results.

3. SOC ARCHITECTURE AND TOOLS

This work targets SoCs employing a bus to connect a processor to memory and IP cores. The proposed approach is prototyped in FPGAs using the Excalibur™ environment[12]. Figure 1 presents the target SoC architecture. A PC is used as the external tester, responsible for loading the test program and deterministic patterns into the memory through the serial interface.

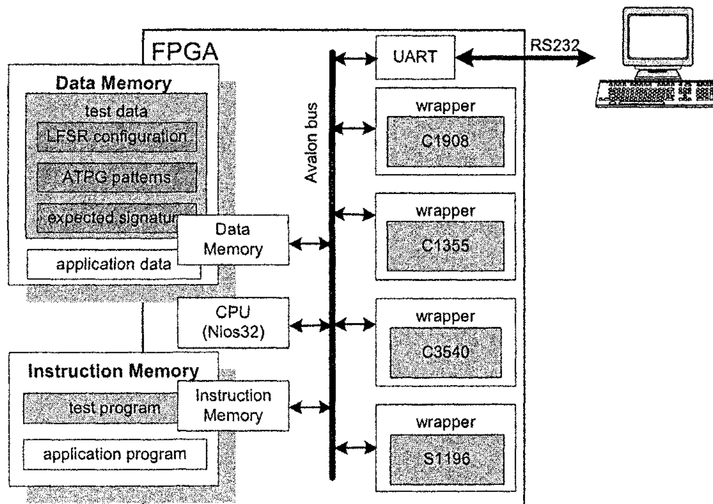

Figure 1. Target SoC architecture.

The embedded processor used is the 5-stage pipeline Nios32[12]. The 32-bit Avalon™ bus is used to connect the SoC components. The c1908, c1355, c3540 and s1196 benchmarks, randomly chosen, are used to evaluate the proposed test method and the CAT tool. Each core connected to the bus is wrapped. Using a uniform wrapper, the same test procedures can be applied to all cores under test.

In the data memory, each core has its LFSR configuration (i.e. the multiple seeds and polynomials) chosen in a manual process, the ATPG patterns generated by a commercial tool, the expected signatures and the application data. The instruction memory contains the synthesized test program used to test all non-programmable cores and the application program.

4. CAT TOOL ENVIRONMENT

The CAT environment developed has four main functions: (i) automatically synthesize test programs in C language; (ii) automatically synthesize the VHDL wrappers to each core; (iii) insert scan chains when necessary; (iv) automatically synthesize the SoC interface to integrate the cores. Figure 2 and Figure 4 present a design flow divided in two parts: the SoC generation and the test program generation.

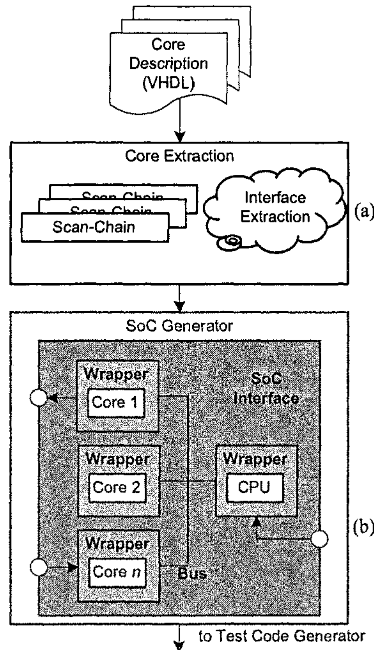

Figure 2. SoC generation CAD flow.

QT (http://www.trolltech.com) is used to build the graphical interface of this environment. The CAT tool synthesizes 'C' and 'VHDL' descriptions from pre-validated templates, integrates commercial CAD tools (Leonardo, ModelSim, FlexTest, FastScan, DFTAdvisor) and guides the user through the design and test flow.

4.1 SoC Generation

The SoC generation flow starts with a tool called *core extraction* –Figure 2(a). This tool extracts the core interface (input/outputs) and may insert scan chains when required. Scan chains are inserted using the DFT Advisor tool. In the interface extraction, some of the extracted data are: (i) the name of the core; (ii) the type of the core - *core*, *processor* or *memory*; (iii) the files composing the core; (iv) the top entity; (v) the ports and generics of the top entity. The user must specify the port usage for each port of the core. Port usage means how the port is used in the system (e.g. *data, clock, reset, enable*, *external*). This is a key piece of information, since it enables the *SoC generator* tool to automatically synthesize the wrapper for this core and the SoC interface.

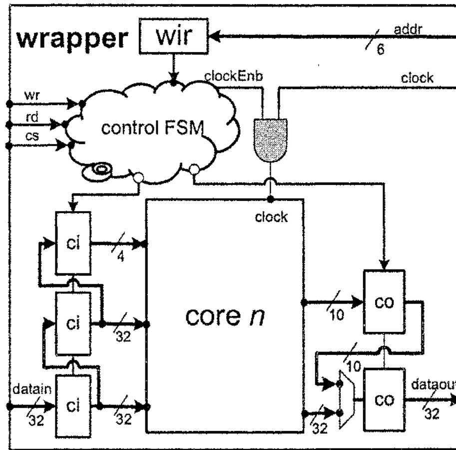

Figure 3. Wrapper scheme.

The next step is the SoC generation, Figure 2(b). The following actions are executed during this step: (i) selection of the cores and the processor; (ii) selection of the communication media; (iii) wrapper synthesis for each core; (iv) SoC interface synthesis.

The cores are selected from a library created in the first step (*core extraction*). After that, the user defines the communication architecture. Different communication architectures can be easily integrated to the environment, but presently only bus-based is allowed.

Once the communication architecture is defined, the wrapper is automatically synthesized for each core. Wrappers are synthesized after the communication architecture selection since the communication protocol and the wrapper interface are associated to the communication media, in this case, a bus.

Figure 3 presents an example of wrapper for a bus-based architecture. The wrapper has shift-registers adapting the bus width (32-bit) to the core input/output widths, represented by *ci* and *co* registers. The test is executed in two steps: patterns loading and response reading. The six least significant bits of the *addr* port are used to set the test mode, i.e. pattern loading or response reading.

When the wrapper is in test mode and a write operation is required (*wr* port), the internal *clockEnb* is asserted during one clock cycle to process the incoming test patterns. Note that the *clockEnb* assertion occurs only when the last part of the incoming pattern is written into the input shift-register.

The responses are stored into the output shift-registers one clock cycle after *clockEnb* assertion. The processor reads the responses when the *rd* port is asserted.

At this step of the flow, two VHDL files are created for each core: *wrapper* and *testbench*. The *testbench* is used to validate the *wrapper*. The test patterns for this *testbench* can be internally generated from a configurable LFSR or read from patterns stored in an external file. When read from external files, it can be used to validate the test patterns and generate the expected signature.

The last step is the SoC interface generation, which is obtained from the data extracted from each core. When the port usage is specified as external, it is connected to the SoC interface. The wrapped core has ports connected to the communication interface and external ports connected to the SoC interface.

These external ports impose modifications (addition of ports) in the wrapper and in the SoC interface.

4.2 Test Program Generation

Once the SoC is built, the next step is to create the *test program*. The test program generates the test vectors, compacts the responses and evaluates the signature of each core. The selected test pattern generation approach is based on Hellebrand's approach[10], which uses different LFSR configurations to increase the fault coverage obtained from pseudo-random patterns. Therefore, fewer deterministic patterns are stored in the memory. The main advantage of this approach is the minimization of technological requirements of the external tester, such as bandwidth, memory and frequency. However, the challenge is to find a tradeoff between run-time for pseudo-random test and the memory requirements for deterministic test. The test program generation flow is presented in Figure 4.

The first action of the *test program generator* (Figure 4(a)) is to select the polynomials and seeds to generate patterns to each core. Multiple polynomials can be selected and each polynomial may have several seeds. The environment supports generic modular LFSR and MISR descriptions. A reseeding tool like[13] could present better encoding efficiency, i.e. more compact test pattern set to achieve higher fault coverage. However, the LFSR configuration was randomly chosen since this kind of tool is not available, at the moment.

Each LFSR configuration is simulated, creating the test patterns for each core - Figure 4(b). The generated patterns are translated to the fault simulator format (Flextest™/Fastscan™) and the fault coverage is evaluated, using the synthesized description of the cores. If the resulting fault coverage is not

sufficient, the user may run the ATPG tool to generate the remaining test patterns or choose other LFSR configuration, restarting the process. At the end of this process, the user has all the necessary patterns to test each core. Thus, the expected signature can be generated using a logic simulator - Figure 4(c).

Finally, these patterns (pseudo-random and/or deterministic) are translated to the test program in C language - Figure 4(d). After the program generation, the user can evaluate the required test time using a hardware/software co-simulation tool - Figure 4 (e). The co-simulation tool is provided by the Excalibur environment. Figure 5 presents an example of pseudo test program.

The core under test in Figure 5 is memory mapped, which means that the processor accesses cores with load and store instructions. This can be observed by the **cutPtr* pointer, in the third line.

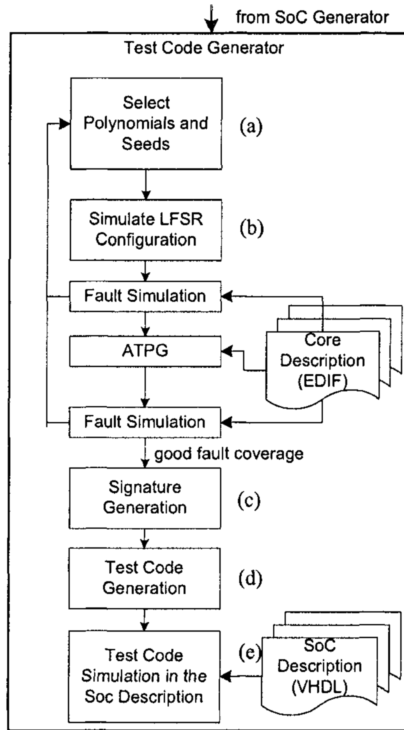

Figure 4. Test program generation CAT flow.

```
1) #define nCores 5
2) const coreTyp coreCfg[nCores]={{expectedSig,corePtr},{},{}};
3) const unsigned detVet[nCores]={{},{},{},{},{}};
4) const unsigned lfsrCfg[nCores]={{},{},{},{},{}};
5) void lfsr( unsigned polynomial, int *state);
6)
7) void ApplyPseudoRandomPattern(volatile void *hwif){
8)    for each polynomial {
9)          poly = lfsrCfg[ind];
10)         n_seed = lfsrCfg[ind+1];
11)         ind+=2;
12)         for each seed {
13)               value = lfsrCfg[ind];
14)               n_patterns = lfsrCfg[ind+1];
15)               ind+=2;
16)               for each pattern {
17)               *hwif = value;      // send pattern to CUT
18)               lfsr(poly;&value);  // generate new pattern
19) }}}}
20) void ApplyDeterministicPattern(volatile void *hwif){
21)    for each pattern {
22)          *hwif = detVet[i]; // send det. pattern to CUT
23) }}
24) int ExecuteTest(void){
25)    volatile void *cut;
26)    for each core {
27)          cut = coreCfg[core].cutPtr;
28)          ApplyPseudoRandomPattern(cut);
29)          ApplyDeterministicPattern(cut);
30)          genSignature = *cut;  // read signature from the
31)                                // CUT - MISR in hardware
32)          if (genSignature != coreCfg[core].expectedSig)
33)                abortTest();
34)    }
35) }}
```

Figure 5. Pseudo test program example. LFSR implemented in software (lfsr function) and MISR in hardware.

The first lines define the expected signature, the deterministic test vectors generated by the ATPG and the LFSR configuration. The *ExecuteTest* procedure is the main test function. For all cores of the system this function applies the pseudo-random patterns and the deterministic patterns. Depending on the test time requirements, the test designer may implement the MISR in hardware (line 28) to speed-up the test, as presented in this example. Each core has an embedded MISR, which is read by the software at the end of the core test. The returned signature is compared against the expected one (lines 30-31). When the test code is generated to each core, the system (hardware and software) is prototyped on a development board for the final validation.

5. RESULTS AND DISCUSSION

The features evaluated using the ISCAS85/89 benchmarks are fault coverage, test time and memory requirements to store the test program (instruction memory) and deterministic patterns (data memory). Table 1 presents some additional information of the benchmarks.

Table 1. Benchmarks general information.

Core	Inputs	Outputs	Core area	Core + Wrapper Area
c1908	33	25	286	843
c1355	41	32	271	948
c3540	50	22	756	1499
s1196	14	14	476	854

The benchmarks are mapped to a 0.35 library and the area is presented in nand2 equivalent gates. The cores evaluated are small and do not represent real case cores. Using complex benchmarks, the wrapper overhead will be reduced since it is a function mainly of the number of inputs and outputs.

Table 2 presents the fault coverage (FC) obtained and the number of patterns for the mixed patterns approach. For example, core c1908 obtained 67.94% fault coverage using only pseudo-random patterns. Deterministic patterns, generated by a commercial ATPG, increased the fault coverage to 99.52%. The last two columns present the number of patterns (pseudo-random and deterministic) used.

Table 3 presents the data memory requirements to store the LFSR configuration (*random size*) and the ATPG patterns (*deterministic size*) in bytes. As can be observed, the number of bytes to store a LFSR configuration is much smaller than deterministic patterns, as stated in [10]. However, a significant number of deterministic patterns had to be used to reach acceptable fault coverage. Those remaining deterministic patterns can be reduced even more using embedded compression algorithms or using a tool to select the best seeds and polynomials for each core. Using compression algorithms will obviously reduce the test data and the time to download them. However, this can have a negative impact in the test time, since the patterns had to be decompressed before being sent to the core. On the other hand, using a tool to select seeds can increase the quality of the pseudo random patterns (i.e. increase the number of useful patterns) reducing the need of deterministic patterns[13].

The mixed test pattern approach (i.e. deterministic and pseudo-random) was partitioned into four different hardware/software test configurations. In the first partition, the LFSR, the MISR and the control structure used to apply deterministic patterns are implemented in hardware (*hardware partition*). In the second partition, all test modules are implemented in software (*software partition*). The third partition implements the LFSR in software and the MISR in hardware (*misr hardware partition*). In the fourth partition, the test patterns are generated using arithmetic functions (*adders software partition)* provided by the processor and the MISR is implemented in hardware[11]. The fault coverage using adders is not evaluated.

Table 2. Fault coverage (FC) and number of test patterns for mixed test patterns approach.

Core	Random FC	Deterministic + Random FC	# Random Patterns	# Deterministic Patterns
c1908	67.94	99.52	40	68
c1355	88.89	100.00	66	29
c3540	75.22	97.48	105	184
s1196	53.41	94.57	159	459

Table 3. Data memory requirements, in bytes, for each core.

Core	Random Size	Deterministic Size
c1908	9	137
c1355	17	59
c3540	21	369
s1196	15	460

Table 4 compares the instruction memory requirements to store the test program (each instruction uses two bytes) for three partitions. The test program is the same for all cores. As can be noted, since the test program is simplified (i.e. moved from LFSR to adder to generate patterns) the test program obviously is reduced in size. As can be seen in Table 4 the instruction memory requirements, i.e. bytes necessary to store the test program, are relatively small. The instruction memory requires less than 1Kbyte to store a test program to test all the non-programmable cores. This shows that the memory requirements to store the test program are not a constraint for software-based test applied to non-programmable core.

Table 5 presents the number of clock cycles between the generation of two patterns. In the hardware partition, the pseudo-random patterns are generated cycle by cycle and the deterministic patterns are estimated to 10 clock cycles, due to the time to access the test memory. The software partition takes 185 cycles to apply a pseudo-random pattern and 175 cycles to apply a deterministic pattern, due to the test serialization. Since part of the test has been removed from the software (i.e. response compaction moved to hardware), the time is reduced. In the last partition, it is possible to see that the test time is even more reduced, confirming[4,11], which suggest the use of specific test instructions to reduce the test time and test program length.

Table 6 complements the test time information presented in Table 5. It presents the total test time for the mixed test approach for the benchmarks using the four partitions.

As expected, the software partition takes longer test time to complete the test. This difference is due to the overhead induced by the data transfer from the processor to the core over the bus, and the test serialization. In the hardware approach, these steps are executed in parallel, as a pipeline structure.

Table 4. Instruction memory requirements, in bytes, for the hardware/software partitions.

Core	Software	LFSR in Software MISR in Hardware	Adder in Software MISR in Hardware
All	564	428	394

Table 5. Time in clock cycles to apply a new pattern to the UUT.

Pattern Type	Hardware	Software	LFSR software MISR hardware	Adder software MISR hardware
Pseudo-random	1	185	102	73
Deterministic	10	175	43	43

Table 6. Total test time, in clock cycles, for four test approach.

Core	Hardware	Software	LFSR software MISR hardware	Adder software MISR hardware
c1355	1400	40168	28034	18817
c1908	646	35313	23246	17552
c3540	3785	94232	61906	46790
s1196	4749	110046	53452	48723

The third and the fourth test partitions reduce the total test time. The MISR implemented in hardware reduces the traffic in the bus, since the responses are compacted in hardware, sending back to the processor only the signature (see the test program in Figure 5). The test time in the last partition was reduced by using a processor specific instruction to generate the pseudo-random patterns. This has an advantage over the LFSR emulation because a new pattern is created with only one instruction, reducing the test time and the test program length[4,11]. The discussion of the results is presented in the next Section.

6. CONCLUSIONS

Software-based test minimizes the two main drawbacks related to BIST: performance degradation and additional hardware area. According to the literature, software-based test increases test time and the test memory requirements. This work presents a quantitative evaluation of these drawbacks. Actually, the test time is an important drawback for manufacturing test, since the test time using only software-based test is much longer compared to the hardware-based test. As shown, the test time is minimized when the test modules are partitioned into hardware/software modules. The test time is also minimized using an adder to generate patterns. On the other hand, the instruction memory requirement for software-based test is low. A careful design of the test program can enable its reuse to test different cores. It is also possible to reduce the data memory requirement using compression algorithms in the deterministic patterns, an issue not

explored in this work. Thus, software-based test can be applied to field test since the memory requirement, the main bottleneck to this kind of test, is minimal.

The added area by the wrapper insertion, Table 1, is used to integrate the cores to the bus, without test-related circuitry. The only area overhead induced by the software-based test is the memory space used to store the test program.

The second contribution of this work is the developed CAT environment, integrating software-based test to a generic and simple to use SoC design flow.

As ongoing work, the CAT environment is being validated with ITC02 benchmark. Other criterions, such as hardware overhead and power dissipation are also being evaluated. We are also evaluating software based-test in other communication architectures like network-on-chip[14].

ACKNOWLEDGEMENTS

Fernando Moraes gratefully acknowledges the support of the CNPq through research grant number 307665/2003-2.

7. REFERENCES

1. L. Chen, and S. Dey, Software-Based Self-Testing Methodology for Processor Cores. *IEEE Transactions on Computer-Aided Designs.* **20**(3), 369-380 (2001).
2. J.-R. Huang, M. K. Iyer, and K.-T. Cheng, A Self-Test Methodology for IP Cores in Bus-Based Programmable SoCs. In: *VLSI Test Symposium.* 198-203 (2001).
3. S. Hwang, and J. A. Abraham, Reuse of Addressable System Bus for SOC Testing. In: *ASIC/SOC Conference.* 215-219 (2001).
4. W. C. Lai, and K. T. Cheng, Instruction-Level DFT for Testing Processor and IP Cores in System-on-a-Chip. In: *Design Automation Conference.* 59–64 (2001).
5. A. Krstic, W. C. Lai, K. T. Cheng, L. Chen, and S. Dey, Embedded Software-Based Self-Test for Programmable Core-Based Designs. *IEEE Design and Test of Computers.* **19**(4), 18–27 (2002).
6. S. M. Thatte, and J. A. Abraham, A Methodology for Functional Level Testing of Microprocessors. In: *International Symposium on Fault-Tolerant Computing.* 90-95 (1978).
7. N. Kranitis, G. Xenoulis, D. Gizopoulos, A. Paschalis, and Y. Zorian, Low-Cost Software-Based Self-Test of RISC Processor Cores. In: *Design, Automation and Test in Europe Conference.* 164-168 (2003).
8. A. J. Van de Goor, Using March Tests to Test SRAMs. *IEEE Design and Test of Computers.* **10**(1), 8–14 (1993).
9. Y. Zorian, and A. Ivanov, An Effective BIST Scheme for ROM's. *IEEE Transaction on Computers.* **41**(5), 646-653 (1992).

10. S. Hellebrand, H. J. Wunderlich, and A. Hertwig, Mixed-Mode BIST Using Embedded Processors. In: *International Test Conference.* 195-204 (1996).
11. J. Rajski, and J. Tyszer, *Arithmetic Built-In Self-Test for Embedded Systems* (Prentice Hall, Upper Saddle River, 1998).
12. Altera Inc. Nios Embedded Processor: 32 bits Programmer's Reference Manual. (version 2.1, 2002, 124 p).
13. C. V. Krishna, and N. A. Touba, Reducing Test Data Volume Using LFSR Reseeding with Seed Compression. In: *International Test Conference.* 321-330 (2002).
14. A. M. Amory, E. Cota, M. S. Lubaszewski, and F. G. Moraes, Reducing Test Time with Processor Reuse in Network-on-Chip Based Systems. In: *Symposium on Integrated Circuits and System.* 111-116 (2004).