Computing and Informatics, Vol. 28, 2009, 97–113

FUNCTIONAL TESTING OF PROCESSOR CORES IN FPGA-BASED APPLICATIONS

Mariusz Wegrzyn, Franc Novak, Anton Biasizzo

Computer Systems Department
Jožef Stefan Institute
Jamova cesta 39
1000 Ljubljana, Slovenia
e-mail: {mariusz.wegrzyn, franc.novak, anton.biasizzo}@ijs.si

Michel Renovell

Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier
161 rue Ada
34392 Montpellier Cedex 5, France
e-mail: renovell@lirmm.fr

Manuscript received 13 November 2007; revised 5 May 2008
Communicated by Elena Gramatová

Abstract. Embedded processor cores, which are widely used in SRAM-based FPGA applications, are candidates for SEU (Single Event Upset)-induced faults and need to be tested occasionally during system exploitation. Verifying a processor core is a difficult task, due to its complexity and the lack of user knowledge about the core-implementation details. In user applications, processor cores are normally tested by executing some kind of functional test in which the individual processor's instructions are tested with a set of deterministic test patterns, and the results are then compared with the stored reference values. For practical reasons the number of test patterns and corresponding results is usually small, which inherently leads to low fault coverage. In this paper we develop a concept that combines the whole instruction-set test into a compact test sequence, which can then be repeated with different input test patterns. This improves the fault coverage considerably with no additional memory requirements.

Keywords: Built-in self-test, embedded processor core test, fault injection, fault modelling, functional test, single-event upset

1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are becoming a widely accepted design style for low- and medium-volume applications and represent a cost-effective alternative to traditional fixed-logic ASICs. Their low development costs and inherent flexibility to change the function performed by the field reconfiguration have resulted in a rapid growth of this technology.

Testing FPGAs requires different solutions to those applicable for ASICs. Programmable devices are composed of components such as complex logic blocks with look-up tables (LUTs), multiplexers and flip-flops, embedded memories, and dedicated routing logic. Production-test techniques concentrate on testing the individual types of functional blocks and their interconnections. The device is programmed with a number of test configurations and specific test stimuli are applied for each test configuration. Since the time spent programming the device with each configuration is several milliseconds, the goal is to test all the functional blocks and routing resources with the minimum test configurations. Different approaches have been proposed for testing FPGA logic blocks [10, 23, 25], FPGA routing resources [7, 18, 21, 22] and detecting delay faults [1, 9]. Since FPGA circuit resources are not normally 100 % occupied by the design, the defects located in some areas of the chip that are not used by a particular design may be tolerated. Hence, a strategy of testing the resources of an FPGA with respect to a specific design to be implemented on it has been proposed. This type of test is referred to as an application-oriented test [26, 27].

In this paper we focus on the application-oriented testing of embedded processor cores implemented in SRAM-based FPGAs, which are relatively sensitive to Single Event Effects (SEEs). SEEs occur when charged particles hit the silicon, transferring enough energy to provoke a fault in the system. SEEs can manifest themselves as permanent or transient faults. The transient effect, also called a Single Event Upset (SEU), results in bit flips in the memory elements. A number of papers on SEU-induced errors in microelectronic circuits have been published, among them [12, 13, 17, 19], and fault-tolerance techniques for SRAM-based FPGAs have recently been published in a book by Kastensmidt et al. [11]. Our goal was to provide a compact and efficient test solution that is suitable for built-in self-test implementations.

In the following we briefly review the main approaches to the software-based self-testing of processor cores. Then we describe our approach based on a data-sensitive path. The approach is illustrated by an experimental case study and evaluated by simulating faults in the HDL (hardware description language) description of the processor core. The achieved fault coverage is reported and compared with other techniques, and in the final section we draw some conclusions.

2 SOFTWARE-BASED TEST TECHNIQUES FOR PROCESSOR CORES

The testing of a deeply embedded processor core with poor accessibility is normally based on a built-in self-test rather than on a conventional functional test with an external ATE because of the communication bottleneck between the processor core and the ATE. In this approach, both the generation of the test pattern and the evaluation of the test results are performed by the processor under test.

Two main approaches to software-based self-testing have been reported: structural and functional. In a structural self-test [4], the test-pattern sequences are developed for each processor component based on a gate-level netlist of the individual core components. Since the gate-level details of processor cores are in most cases not available to the designer, for reasons relating to the protection of intellectual property, this imposes serious restrictions on the test's application in practice. A high-level structural self-test methodology [14, 15] tries to overcome this problem in the sense that it is based on knowledge of the Instruction Set Architecture (ISA) of the processor and its Register Transfer (RT) level description. The RT level description of the functional parts of the processor and their interconnections are more likely to be available to users (although problems still remain in evaluating the efficiency of the implemented self-test if the corresponding tools for fault simulation are not provided for the target processor core).

In a functional self-test the processor cores are tested by executing a sequence of instructions that exercise the functional behaviour of the processor. The design of the functional self-test is related to a functional description of the processor's instructions. In earlier implementations, individual processor's instructions are tested with a set of deterministic test patterns and the results are compared with the stored reference values. For practical reasons the number of test patterns and the corresponding number of results is usually small, which inherently leads to a low fault coverage. In [24], random instruction sequences are generated for testing individual instructions. Furthermore, the results are compressed by means of a signature analyser incorporated in the functional test software. This approach can be applied both to manufacture testing and post-manufacture self-testing. Its drawback during self-testing is the excessively large size of the test code due to the use of a pseudorandom strategy.

An alternative approach, called the instruction randomization self-test (IRST), reported in [2], relies on pseudorandom operations and operands of instructions. The instruction randomization is performed with dedicated hardware, which modifies certain instruction fields such that the instruction remains meaningful and its operands are randomly permutated. In this way the operation of the processor is explored in many more situations, which increases the fault coverage. This approach, however, suffers from the same drawback as in the previous case.

Recent approaches [3, 5, 6] perform the processor test with a compact sequence of instructions, trying to achieve a high stuck-at fault coverage by the proper selection of instructions and instruction operands. In [5, 6] the test sequence is generated by a genetic algorithm. In this approach the test-program generator produces test programs by inducing an external instruction library that describes the syntax of the microprocessor assembly language. The generator utilizes a fault simulator to evaluate the generated test programs. The test-program generation and evaluation are performed in consecutive steps of a genetic algorithm until the required fault coverage is achieved. Different strategies for generating the test sequence for a target i8051 processor are reported in [5], with the stuck-at fault coverage varying from about 36 to 91 %. In [6] the approach is generalized using the feedback information from a simulator that makes possible an evaluation with respect to particular coverage metrics. The reported case study is performed on a Sparc V8, a relatively complex microprocessor with a five-stage pipelined architecture.

The above techniques primarily focus on efficient test-sequence generation. Their integration into the self-test of an embedded processor core remains an open issue.

3 SENSITIVE-PATH APPROACH

3.1 Basic Principle

In our approach the goal is to generate a compact test sequence that detects permanent SEU-induced faults of embedded processor cores in SRAM-based FPGA circuits. As described in [20], the functional model of such faults differs considerably from the conventional stuck-at fault model due to the fact that SEU-induced faults affect logic elements implemented by the look-up tables such that the logic function is arbitrarily changed. While the existing fault simulators do not cover such a functional fault model, we follow an implicit strategy of test adequacy and statistical testing [16]. As such we generate a test sequence that allows arbitrary situations that might occur in practice and consequently detects faults that only appear in a particular sequence of events. This is accomplished by using a test sequence that explores the functionality of each individual instruction and is composed in such a way that it forms a sensitive path, which can be executed more than once, each time with a different input pattern.

Although we borrow the notion of a sensitive path from the automatic test-pattern-generation (ATPG) techniques [8], in our case it has a slightly different meaning. The path sensitization in conventional ATPG techniques for automatic test generation involves the generation of a path that is sensitive to the presence of a stuck-at fault and the justification of the values along the path by propagating signals back to the primary inputs. In our case the fault detection is performed at the instruction level by a compact test program in which individual processor instructions are organised in a sequence such that the destination register operand of instruction i represents the source register operand of instruction $i + 1$. In the test sequence, each processor instruction participates at least once (in order to test the instruction decoder of the processor core). Intuitively we assume that the test sequence represents a sensitive path if the data flow through it is sensitive to changes of the input pattern. We pursue the following two goals:

- the faults occurring during the execution of individual instructions in the test sequence should manifest themselves in the final result,
- the data-sensitive path should provide a way of randomizing the instruction operands of the test sequence, resulting in increased processor activity and consequently in increased fault coverage.

A data-sensitive path can be achieved by following the two basic principles of design-for-testability: controllability and observability. Controllability is the ability to set the values of the inputs of any system component from the primary inputs of the system. Observability is the ability to observe the values of the outputs of any system component at the primary outputs of the system. An instruction of a test sequence can be regarded as a system component. The test sequence is composed of individual instructions (i.e., system components), which act upon the data stored in registers and memory cells. An instruction processes the input data (i.e., the argument) and generates a result that represents the input data of the next instruction in the test sequence. The input data of the first instruction of the test sequence represents the system's primary inputs, while the results of the test sequence system are the primary outputs. The test sequence is composed in an incremental way: each time a new instruction is added to the test sequence the resulting test block is checked for controllability and observability.

3.2 Bijective Property

The controllability and observability principle is only a vague concept that leads to different implementations of the test sequence with a relatively diverse fault coverage. Instead of introducing some kind of metrics as a guideline to efficient solutions, we impose a stricter rule on the test-sequence generation by requiring that there is a one-to-one, i.e., bijective, correspondence between the input test pattern and the result. If we apply this rule at the level of sub-sequences of the instruction sequence we can ensure high controllability and observability within the whole instruction sequence, which is a prerequisite for achieving high fault coverage.

3.3 Refinements

For some instructions the output data may not be completely sensitive to the changes of input data and hence the property of a sensitive data path is not preserved. For example, some part of the register holding the result of the instruction operation may be cleared or set to all 1's. Additional data manipulation needs to be performed (i.e., the input data is stored at another location and logically combined with the result of the executed instruction).

The execution of some instructions affects the status flags (like, for example, the zero flag, carry, etc.) In order to detect possible faults in the status information, the contents of the status register are included in the result of the currently executed instruction. This is normally done by XORing the contents of the status register and the resulting output data.

With such refinements the instruction and additional data manipulation code represent a bijective block within the test sequence.

For illustration, a part of the test sequence organized in a data-sensitive path is shown in Figure 1. The destination register operand of an instruction represents the source register operand of the next instruction in the test sequence. Test sequence is composed of bijective blocks. A bijective block can be a single instruction if it exhibits bijective property. If not, some additional data manipulation is required to obtain a bijective block.

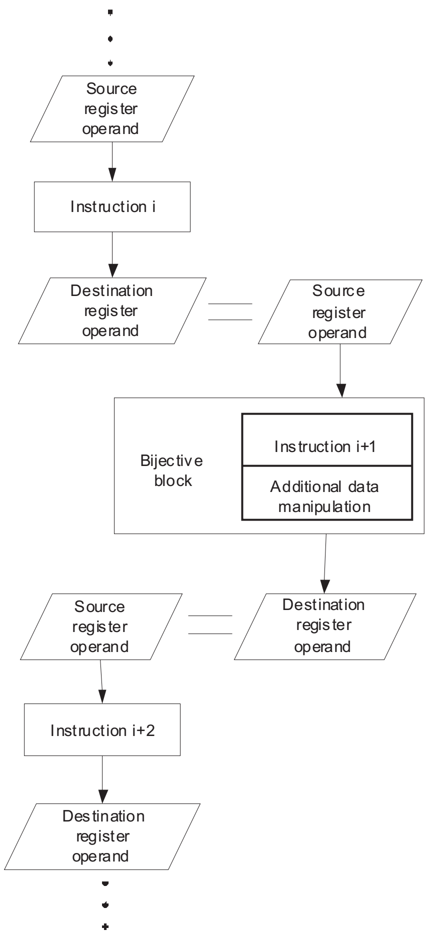

Fig. 1. Test sequence organised in a data-sensitive path with k iterations

3.4 BIST Implementation

A common test strategy aims at minimizing test time and test overhead. In our case the two goals are to some extent contradictory, as will be shown in the following.

A general test situation can be described by:

- the set of faults $F = \{f_1, f_2, \ldots, f_m\}$;
- the set of available binary tests $T = \{t_1, t_2, \ldots, t_n\}$ where t_i corresponds to the execution of the test sequence with input test pattern i $(1 \leq i \leq n)$;
- the set of results $R = \{r_1, r_2, \ldots, r_n\}$ where r_i is the result of the execution of the test t_i;
- the test matrix D describing test capabilities of tests T. Each test t_j, $1 \leq j \leq n$ is related to a binary test vector $d_j = [d_{0j}, d_{1j}, \ldots, d_{mj}]$. The value $d_{ij} = 1$ denotes that test t_j detects fault f_i. Conversely, $d_{ij} = 0$ indicates that t_j does not detect f_i. Binary test matrix D consists of test vectors $D = [d_1, d : 2, \ldots, d_n]$. Its dimension is $m \times n$.

The test strategy that minimizes test time is as follows:

```
begin
    test_ queue is empty;
    remaining_faults is $F$;
    while remaining_faults is not empty
        begin
            determine the most hard-to-detect fault $f_i$;
            select test $t_j$ that detects fault $f_i$;
            put $t_j$ in the test_ queue;
            omit all faults detected by $t_j$ from remaining_faults;
        end
end
```

The most hard-to-detect fault f_i is determined from the test matrix D as the one detected by the least number of tests. BIST implementation requires memory resources for storing the resulting test queue (i.e., input test patterns), the instruction test sequence and the test results.

Alternatively, the result of the previous execution of the test sequence can serve as the input test pattern for the next execution of the test sequence. This is possible if the consecutive results used as input test patterns form one or more cyclic groups. The selection of the shortest test-pattern sequence of a cyclic group that achieves maximum fault coverage is a difficult optimization problem, which we do not address in this paper. Instead we accept a sub-optimal practical solution by selecting the initial test pattern that detects the most hard-to-detect faults and execute the test sequence with k subsequent test patterns striving to achieve the target fault coverage.

The test strategy that minimizes test resources is as follows:

```
begin
    test_queue is empty;
    remaining_faults is $F$;
    determine the most hard-to-detect fault $f_i$;
    select test $t_j$ that detects fault $f_i$;
    put $t_j$ in the test_queue;
    omit all faults detected by $t_j$ from remaining_faults;
    while remaining_faults is not empty
        begin
            put the result $r_j$ of the last test $t_j$ in the test_queue;
            omit all faults detected by $t_j$ from remaining_faults;
        end
end
```

Notice that the above algorithm assumes that it is possible to reach the maximum fault coverage with test patterns in a given cycle group. Hence only the initial test pattern, the number of executions of the test sequence and the result are stored. If this is not the case, test queue is extended over more than one cycle group, and proportionally more input test patterns and test results need to be stored.

4 ILLUSTRATIVE CASE STUDY

When selecting the processor core for the case study, an important question to ask is how will the developed solution actually be evaluated? In order to determine the fault coverage of SEU-induced faults some means of fault injection must be provided. As an alternative to the statistics-based radiation tests we are looking for a simulation-based solution. Simulation-based fault injection is made difficult by the lack of commercial tools that would allow the user to alter the FPGA configuration once it is translated from the HDL source into the target FPGA platform. However, one possible solution is to use the debug option to insert faults during the test program's execution. This approach is general (i.e., it can be applied for different processor cores), but it also has some limitations and requires rather a lot of manual interference. Alternatively, the HDL description can be used to model the system and simulate its performance as well as the faults within the system.

We evaluated our approach on a Xilinx PicoBlaze processor core for which the low-level HDL source code is available. The Xilinx PicoBlaze processor is a small 8-bit microprocessor, used mainly for training purposes. It has 1K of program space, 16 8-bit registers, 256 input and 256 output ports, a 64-byte internal scratchpad RAM and a 31-location stack. Since the Xilinx PicoBlaze processor core is designed for FPGA implementation its HDL description consists of low-level FPGA functional blocks that are directly mapped to the FPGA resources. The behaviour of the system can be viably simulated on any HDL simulator supplied with Xilinx library primitives. On the other hand, this low-level HDL description allows us to inject faults and simulate the system responses relatively easily.

4.1 Assumed Fault Modelling and Details of the Fault-Injection Process

The PicoBlaze processor core was mapped into a Virtex family FPGA from Xilinx. The Virtex's architecture consists of a regular structure of tiles of configurable logic blocks (CLB) surrounded by programmable input/output blocks and routing resources. The tile of a configurable logic block is composed of CLB slices, three-state buffers, input and output multiplexers and a routing matrix. A CLB slice consists of LUTs, flip-flops and multiplexers. A LUT is merely a static random-access memory that makes it possible to implement individual combinatorial functions. When programming the FPGA, the configuration memory cells associated with individual LUTs, flip-flops, CLB configuration cells and interconnections are loaded. In a given programmed configuration, the LUT's contents implement the target-function truth table.

Although the single upset event (SEU) is a transient effect it can manifest itself as a permanent fault in a SRAM-based FPGA. This is due to the fact that the majority of the FPGA SRAM is used for the FPGA configuration matrix and a single-bit flip in the configuration matrix can modify the FPGA's functionality. All the bits in the configuration memory cells described above are potentially sensitive to SEU-induced faults.

```
or_LUT1: LUT3
  generic map (INIT => X"FE")
  port map( I0 => logical_result(1),
            I1 => arith_result(1),
```

a)

```
or_LUT1: LUT3
  generic map (INIT => X"7E")
  port map( I0 => logical_result(1),
            I1 => arith_result(1),
```

b)

Fault-Free LUT

I2	I1	I0	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

c)

$\longrightarrow$

MSB bit of LUT is changed

I2	I1	I0	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

d)

Fig. 2. a) HDL description of fault-free three-input OR gate, b) most significant bit of the LUT is changed (X″FE″ $\rightarrow$ X″7E″)

The faults in an HDL description of a system are modelled by modifying the individual functional blocks. For each functional block an HDL model describing the behaviour of SEU-induced faults is developed. The HDL model should actually reflect the change of configuration as a consequence of the SEU effect. The same system can be described in HDL in a number of different ways. The two extreme cases are high-level algorithmic descriptions (irrespective of the target system's structure) and RTL (register transfer logic), a level description in which functionality is described at the level of operations among the actual system components (i.e., registers). A high-level description of a core does not provide enough details for realistic HDL modelling of the SEU-induced faults. On the other hand, soft processors like the Xilinx PicoBlaze are developed for FPGA implementation. Their HDL descriptions reflect the FPGA structure in order to efficiently use the FPGA resources that allow precise modelling of the faults and their automated fault injection. The basic HDL entities in a description of the Xilinx PicoBlaze processor core are RAMs, LUTs, multiplexers and flip-flops. SEU-induced faults can alter the contents of RAMs, LUTs or flip-flops or they can modify the connections between these functional blocks. Our goal is to detect permanent faults in the configuration of the processor core. RAM and flip-flop content changes are of a transient nature and can be modified (i.e., restored to a fault-free value) during normal system operation. These faults are detected with an online functional test, specific to the target application and hence not the subject of this investigation. An example of a modelled fault is shown in Figure 2. The HDL description of a LUT implementing a three-input OR gate is shown in Figure 2 a) and the corresponding truth table, in Figure 2 c). The implemented logic function is defined by the initialization parameter (INIT). The SEU-induced fault of a LUT typically manifests itself as a change of one bit of the LUT, thus modifying the Boolean function it implements. Let us assume that the most significant bit of the LUT has been changed, as shown in Figure 2 d). The fault can be modelled by changing the initialization parameter (INIT), as shown in Figure 2 b).

In a similar way the stuck-at faults at the LUT inputs as well as the stuck-at faults at the LUT output can also be modelled by modifying the contents of the LUT configuration. An example of a stuck-at-0 fault of input I2 is depicted in Figure 3. The contents of the LUT in Figure 3(a) are changed by initializing the parameter (INIT), as shown in Figure 3 b). The truth tables corresponding to the fault-free LUT and the stuck-at-0 fault of input I2 are shown in Figures 3 c) and d).

The fault injection was implemented in two steps:

- a description of the faults,
- an HDL simulation of the system with generated faults.

The generation of the fault descriptions was implemented as a perl script. All the instances of the LUT functional blocks are located in the HDL description of the processor core. For each LUT instance its initialization parameter is investigated and the list of the initialization parameters describing all the SEU-induced faults as well as all the stuck-at faults at the LUT inputs and output is generated. For some LUT instances it is possible that a single bit change of a LUT content manifests

```
or_LUT1: LUT3
   generic map (INIT => X"FE")
   port map( I0 => logical_result(1),
             I1 => arith_result(1),
             I2 => shift_result(1),
             O => alu_group(1));
```

a)

```
or_LUT1: LUT3
   generic map (INIT => X"EE")
   port map( I0 => logical_result(1),
             I1 => arith_result(1),
             I2 => shift_result(1),
             O => alu_group(1));
```

b)

Fault-Free LUT

I2	I1	I0	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1
			X"FE"

c)

I2 stack-at-0

I2	I1	I0	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
			X"EE"

d)

Fig. 3. a) HDL description of fault-free three-input OR gate, b) input I2 stuck-at 0 fault ($X''FE'' \rightarrow X''EE''$)

itself as a stuck-at fault. In such a case a duplicated stuck-at fault description is omitted. During fault simulation the generated "faulty" initialization parameters were applied one by one to the HDL description of the Xilinx PicoBlaze processor core. A modified HDL description was used, running the test sequence with different input patterns and the results were recorded for a later offline evaluation. A Cadence NC VHDL simulator running on a Sun Fire V240 server was used for the HDL simulation. The number of injected faults was 1954 and the simulation time for the total 256 input patterns was about 5 hours.

4.2 Implementation of the PicoBlaze Processor Core Test

A test sequence following the principles described in Section 3 was implemented and executed for all possible input test patterns. In an early phase, an analysis of the results showed that the sequence is not completely bijective. By re-running the test sequence and analysing the data at selected points within the data path we spotted the non-bijective sub-parts and modified them. For an illustration consider the subsequence related to the test of the Shift Right Arithmetic (SRA) instruction shown in Figure 4. Since the least significant bit (LSB) of the register is shifted out to the carry flag and at the same time the carry flag is shifted in, the most significant bit (MSB) of the register, the SRA instruction, alone does not preserve the bijective transformation and is dependent on the previous value of the carry flag. In order to implement a bijective transformation the carry flag is initialized to the value of the 4th bit of the register, and after the shift operation the "escaped" bit captured in the carry flag is combined with the value of the shifted register, thus restoring the original value of the register.

```
SRA_TEST :  test  sc, 10 ; set carry flag to 4th bit of SC reg.
            sra   sc     ; shift right arithmetic SC register
            addcy sf, 0  ; restore LSB bit of initial register
            xor   sc, sf ; combining restored LSB bit with
                         ; shifted value without loss of
                         ; information into SC register.
                         ; (initial value can be restored)
```

Fig. 4. Test of Shift Right Arithmetic (SRA) instruction

4.3 Experimental Results

For a comparison of the achieved fault coverage, some other programs that exploit the functionality of the processor core (like in [5]) have been implemented. The results are given in Table 1.

Test program *fibonacci* is a conventional implementation of Fibonacci series, while *fibonacci (recursion)* is a recursive version which exploits some additional resources (i.e., stack) as can be seen from the increased fault coverage. Both test programs have low fault coverage because they do not explore the complete set of processor instructions. Test program *random instruction order* includes the complete instruction set but controllability and observability have not been respected in individual implementation steps. The last two test programs are composed following the data sensitive path approach. In the first, the test sequence was composed in *ad hoc* way, while in the last bijective property was strictly respected.

test program	Stuck-at faults		complete list	
	934 faults	883 faults	1932 faults	1730 faults
fibonacci	49.9	52.8	33.8	37.7
fibonacci (recursion)	56.0	59.2	43.3	48.4
random instruction order	62.7	66.4	46.8	52.3
data sensitive path (not completely bijective)	72.6	76.8	62.9	70.2
data sensitive path + bijective sub-sequences	88.1	93.2	76.6	85.6

Table 1. Fault coverage (%)

The achieved fault coverage is presented in two column pairs. The first column pair refers to the fault coverage where only stuck-at faults were injected. The second column pair presents the complete fault coverage with both stuck-at faults and functional faults in the LUTs. For each pair, the left-hand column refers to all the simulated faults (i.e., 934 in the case of the stuck-at faults and 1932 in the case of the stuck-at faults and the functional faults in the LUTs). The test sequence does not test some specific types of faults related to input/output operations (e.g., interrupt driven routines). If we neglect these faults, the fault coverage of the remaining faults (i.e., 836 in the case of the stuck-at faults and 1730 in the case of the stuck-at faults and the functional faults in the LUTs) is given in the right-hand column.

The fault coverage for the stuck-at faults is given only for a comparison with other reported solutions. The main interest is focused on the fault coverage of the complete list of the injected faults. The initially achieved fault coverage for the proposed approach was 76.6 %. If we neglect the faults related to the input/output operations we get an 85.6 % fault coverage. An analysis of the remaining 14.4 % of the faults that were not detected showed that 4.8 % corresponded to specific fault situations – their detection requires specific values loaded in given registers plus specific values of status flags (i.e., zero result, carry) as a result of the execution of previous instructions. Such faults are difficult to detect with the automatic generation of functional tests. In our case we managed to detect half of them by simply interchanging the order of the sub-sequences, the other half required a manual modification of the test sequence. A total of 5.2 % of the faults proved to be due to the redundancy of the implemented logic blocks of the processor core. The remaining 4.4 % require an in-depth analysis and are still the subject of an investigation.

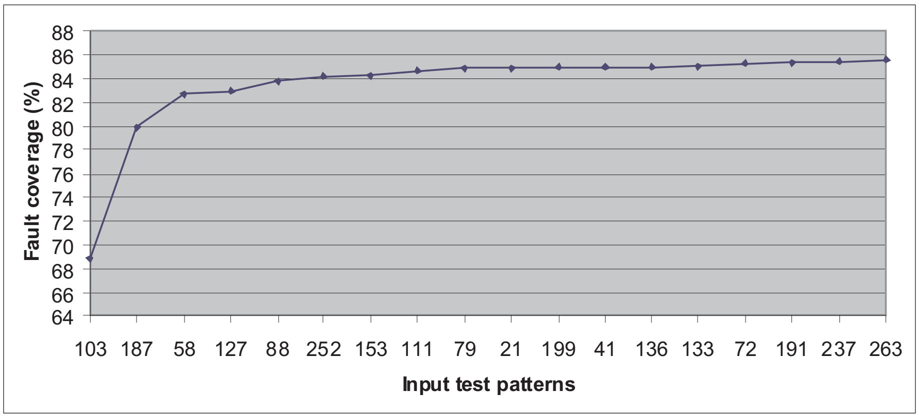

Fig. 5. Achieved fault coverage during the repeated runs of a test sequence

Comparison of the achieved fault coverage among different test programs exhibits the same trends as in [5]. There is, however, a substantial difference in modelled faults: in addition to the conventional stuck-at model we also include functional faults which are more difficult to detect.

As described in Section 3, the test sequence in our approach can be executed with different test strategies. An example of test strategy that minimizes test resources is shown in Figure 5. Running the test sequence with the initial input test pattern 103 gives a 68.8 % fault coverage. Repeating the test sequence with the resulting test pattern 187 increases the fault coverage to 79.9 %. In the next step, with the resulting pattern 058, the fault coverage increases to 82.7 %, etc. In this particular case 18 runs of the test sequence were required to reach the 85.6 % fault coverage.

5 CONCLUSIONS

Our proposed approach of functionally testing processor cores produces compact test sequences that are suitable for built-in self-test implementations. The test sequence, organised in a sensitive data path, can be repeated several times, each time with a different input test pattern, which increases the probability of detecting faults. A relatively high initial fault coverage can be obtained if the bijective rule is applied at the level of sub-sequences of the instruction sequence. In some cases, fault coverage can be improved by interchanging the order of the sub-sequences. The concept of a data-sensitive path with a bijective property can be formalised, which opens up the possibility of an automatic test-sequence generation. This remains the subject of our future work. Modelling the faults in an HDL description and the corresponding fault-injection process is another issue that can be further elaborated. In particular, a well-structured HDL description allows an algorithmic identification of the parts of the code that, when properly modified, model SEU-induced faults. Again, the fault-injection process can be made automatic for individual classes of faults. An evaluation case study of a functional test for the PicoBlaze processor core was performed on a generalised fault model, including both stuck-at faults and functional faults in LUTs, which are more difficult to detect. The achieved fault coverage confirms the efficiency of the proposed approach.

Acknowledgements

This work was partially supported by the French-Slovenian bilateral project, contract number BI-FR07-PROTEUS-016.

REFERENCES

[1] Abramovici, M.—Stroud, C.: BIST-based delay fault testing in FPGAs. In Proceedings of the 8th IEEE International On-Line Testing Workshop, pp. 131–134, 2002.

[2] Batcher, K.—Papachristou, C.: Instruction Randomization Self Test for Processor Cores. In Proceedings of the 17th IEEE VLSI Test Symposium, pp. 34–40, 1999.

[3] Chen, L.—Dey, S.: DEFUSE: A Deterministic Functional Self-Test Methodology for Processors. In Proceedings of the 18th IEEE VLSI Test Symposium, pp. 255–262, 2000.

[4] Chen, L.—Dey, S.: Software-Based Self-Testing Methodology for Processor Cores. IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 20, 2001, No. 3, pp. 369–380.

[5] Corno, F.—Cumani, G.—Sonza Reorda, M.—Squillero, G.: Fully Automatic Test Program Generation for Microprocessor Cores. In Proceedings of the Design Automation & Test in Europe Conference, pp. 1006–1011, 2003.

[6] Corno, F.—Sanches, E.—Sonza Reorda, M.—Squillero, G.: Automatic Test Program Generation: A Case Study. IEEE Design and Test of Computers, Vol. 21, 2004, No. 2, pp. 102–109.

[7] Doumar, A.—Ito, H.: Testing the logic cells and interconnect resources for FPGAs. In Proceedings of the 8th Asian Test Symposium, pp. 369–374, 1999.

[8] Eldred, R. D.: Test Routines Based on Symbolic Logical Statements. Journal of the ACM, Vol. 6, 1959, No. 1, pp. 33–37.

[9] Harris, I.—Menon, P.—Tessier, R.: BIST-Based Delay Path Testing in FPGA Architectures. In Proceedings of the International Test Conference, pp. 932–938, 2001.

[10] Huang W.—Lombardi, F.: An Approach to Testing Programmable/Configurable Field Programmable Gate Arrays. In Proceedings of the 14th IEEE VLSI Test Symposium, pp. 450–455, 1996.

[11] Kastensmidt, F. L.—Carro, L.—Reis, R.: Fault-Tolerance Techniques for SRAM-based FPGAs. Frontiers in Electronic Testing. Springer, 2006.

[12] Katz, R.—LaBel, K.—Wang, J. J.—Cronquist, B.—Koga, R.— Penzin, S.—Swift, G.: Radiation Effects on Current Field Programmable Technologies. IEEE Transactions on Nuclear Science, Vol. 44, 1997, No. 6, pp. 1945–1956.

[13] Katz, R.—Wang, J. J.—Reed, R.—Kleyner, I.—D'Ordine, M.—McCollum, J.—Cronquist, B.—Howard, J.: The Effects of Architecture and Process on the Hardness of Programmable Technologies. IEEE Transactions on Nuclear Science, Vol. 46, 1999, No. 6, pp. 1736–1743.

[14] Kranitis, N.—Gizopoulos, D.—Paschalis, A.—Zorian, Y.: Instruction-Based Self-Testing of Processor Cores. In Proceedings of the 20th IEEE VLSI Test Symposium, pp. 223–228, 2002.

[15] Kranitis, N.—Paschalis, A.—Gizopoulos, D.—Zorian, Y.: Effective Software Self-Test Methodology for Processor Cores. In Proceedings of the Design Automation & Test in Europe Conference, pp. 592–597, 2002.

[16] Kuball, S.—May, J.: Test-Adequacy and Statistical Testing: Combining Different Properties of a Test-Set. In Proceedings of the 15th International Symposium on Software Reliability Engineering, pp. 161–172, 2004.

[17] May, T. C.—Woods, M. H.: Alpha-Particle-Induced Soft Errors in Dynamic Memories. IEEE Transaction on Electron Devices, Vol. 26, 1979, No. 1, pp. 2–9.

[18] Michinishi, H.—Yokohira, T.—Okamoto, T.: A Test Methodology for Interconnect Structures of LUT-Based FPGAs. In Proceedings of the 5th Asian Test Symposium, pp. 68–74, 1996.

[19] Petersen, E. L.—Shapiro, P.—Adams, J. H.—Burke, E. A.: Calculation of Cosmic-Ray Induced Soft Upsets and Scaling in VLSI Devices. IEEE Transactions on Nuclear Science, Vol. 29, 1982, No. 6, pp. 2055–2063.

[20] Rebaudengo, M.—Sonza Reorda, M.—Violante, M.: A New Functional Model for Fpga Application-Oriented Testing. In Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 372–380, 2002.

[21] Renovell, M.—Figueras, J.—Zorian, Y.: Test of RAM-Based FPGA: Methodology and Application to Interconnect. In Proceedings of the 15th IEEE VLSI Test Symposium, pp. 230–237, 1997.

[22] Renovell, M.—Portal, J.—Figueras, J.—Zorian, Y.: Testing the Interconnect of RAM-Based FPGAs. IEEE Design and Test of Computers, Vol. 15, 1998, No. 1, pp. 45–50.

[23] Renovell, M.—Zorian, Y.: Different Experiments in Test Generation for XILINX FPGAs. In Proceedings of the International Test Conference, pp. 854–862, 2000.

[24] Shen, J.—Abraham, J. A.: Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation. In Proceedings of the International Test Conference, pp. 990–999, 1998.

[25] Stroud, C.—Konala, S.—Ping C.—Abramovici, M.: Built-in Self-Test of Logic Blocks in FPGAs (Finally, a Free Lunch: BIST without Overhead). In Proceedings of the 14th IEEE VLSI Test Symposium, pp. 387–392, 1996.

[26] Tahoori, M. B.: Application-Specific Bridging Fault Testing of FPGAs. Journal of Electronic Testing, Theory, and Application, Vol. 20, 2004, No. 3, pp. 279–289.

[27] Tahoori, M. B.—McCluskey, E. J.—Renovell, M.—Faure, P.: A Multi-Configuration Strategy for an Application Dependent Testing of FPGAs. In Proceedings of the 22nd IEEE VLSI Test Symposium, pp. 154–159, 2004.

Mariusz Wegrzyn gained M. Sc. degree at the Faculty of Electronics, Computer Science & Telecommunications, Technical University of Gdansk in 2002 and he is now finishing his Ph. D. degree at Jožef Stefan Institute, Ljubljana.

Franc Novak is Head of the Computer Systems Department at the Jozef Stefan Institute, Ljubljana, and associate professor at the Faculty of Electrical Engineering and Computer Science, University of Maribor. His research interests are in the areas of electronic testing and diagnosis, fault-tolerant computing, and design for testability. He has an M. Sc. (1977) and a Ph. D. (1988) in electrical engineering, both from the University of Ljubljana.

Anton Biasizzo is a researcher at Jožef Stefan Institute since 1991. He received Ph. D. degree from the University in Ljubljana in 1998. His research interests include efficient algorithms for sequential diagnosis, constraint logic programming, model based diagnosis and automatic test pattern generation.

Michel Renovell received his Ph. D. degree in applied physics in 1986 from the University of Montpellier, France. He joined the Laboratory of Computer Science, Automation and Microelectronics of Montpellier (LIRMM) in 1986 where he served as Head of the Microelectronics team from 2000 to 2005. He is currently Associate-Director of the ST2I department at the CNRS (Centre National de la Recherche Scientifique) headquarters managing more than 200 French labs. He is member of the editorial board of JETTA, IEEE Design & Test and VLSI Journal. His research interests include defect-oriented testing, analog testing and FPGA testing.