Mariusz WĘGRZYN, **Janusz SOSNOWSKI**
INSTITUTE OF COMPUTER SCIENCE, WARSAW UNIVERSITY OF TECHNOLOGY,
Nowowiejska 15/19, 00-665 Warsaw

Testing schemes for systems based on FPGA processor cores

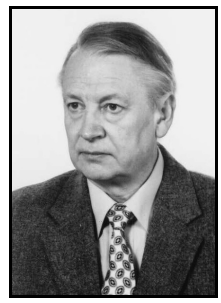
M.Sc. Mariusz WĘGRZYN

He gained M. Sc. degree at the Faculty of Electronics, Computer Science and Telecommunications, Technical University of Gdansk in 2002. He worked as young researcher at Jožef Stefan Institute, Ljubljana in years 2005-2008. Currently he works at Sigarden S.A. in Gdańsk as designer of Computers for special applications FW/HW.

e-mail: marioweg@o2.pl

Prof. Janusz SOSNOWSKI

He is the professor in Institute of Computer Science at the Warsaw University of Technology. He chairs Department of Computer Software and Architecture. He is the author and coauthor of over 200 publications. His research area relates to computer dependability (diagnostics, fault tolerance, reliability), computer architecture and communication interfaces.



e-mail: jss@ii.pw.edu.pl

Abstract

Many systems implemented in FPGAs are based on embedded processor cores (the so called soft cores). Testing such systems is a challenging task due to possible faults in functional blocks, configuration memory and relevant circuitry. The paper deals with software-based self-test schemes taking into account an important requirement on test memory and time overheads. Special attention is paid to configuration faults caused by SEUs (single event upsets). The effectiveness of the proposed method has been verified in fault injection experiments.

Keywords: testing processor cores, application driven testing, FPGA.

Testowanie systemów FPGA wykorzystujących rdzenie procesorów

Streszczenie

W systemach wbudowanych realizowanych na bazie struktur FPGA coraz częściej wykorzystuje się skonfigurowane rdzenie procesorów. Testowanie takich systemów jest dość dużym wyzwaniem ze względu na dość szeroką klasę możliwych błędów w blokach funkcjonalnych, pamięci konfiguracyjnej i związanej z nią logiką. W pracy przedstawiono koncepcje testowania programowego rdzeni procesorów (podejście funkcjonalne, strukturalne, pseudo przypadkowe i aplikacyjne). Szczególna uwagę poświęcono błędom pamięci konfiguracji wynikającym z błędów przemijających, których źródłem jest promieniowanie kosmiczne, szczątkowe promieniowanie użytych materiałów w systemie, czy tez zakłócenia elektryczne. Dokładniej omówiono koncepcje testów złożonych z sekwencji instrukcji, w których wyniki są jednocześnie argumentami wejściowymi dla kolejnych sekwencji (tzw. sekwencje bijektywne). Rozpatrzono problem optymalizacji takich testów biorąc pod uwagę narzut pamięci i czasowy testu oraz pokrycie błędów. Efektywność testów została zweryfikowana w eksperymentach z symulacją błędów. Podane przykłady dotyczą rdzenia procesora 8 bitowego PicoBlaze. Przedstawiona metodyka może być rozszerzona na inne procesory.

Słowa kluczowe: testowanie rdzeni procesorowych, testowanie aplikacyjne, FPGA.

1. Introduction

Recently, appearance of complex FPGAs has allowed implementing quite sophisticated circuitry [4]. In many applications we can simplify design processes by embedding processors into FPGAs and use them in the final application. These processors can also cooperate with specialized embedded circuitry. Many processor cores are available for FPGAs, so they can be easily configured. When developing such systems we face the problem of testing. Classical techniques of testing all FPGA resources are time consuming [5, 6, 7]. In practice, we mostly configure FPGAs to a specific structure or use a preprogrammed microprocessor. In this case universal FPGA testing can be replaced by more effective application driven testing. In the paper we describe such an approach dedicated to an FPGA system based on processor cores. Here, we combine classical testing of fixed logic processors with testing configuration resources of FPGA. In particular, we take into account the problem of detecting configuration memory faults. They relate to cosmic or alpha particle radiation, electrical disturbances, etc. The paper presents an original methodology which has been verified for a simple processor core PicoBlaze, the problem of test optimization is also considered here. Test optimization is targeted at minimalizing requirements for storing initial data test patterns and test results. The effectiveness of the proposed methodology has been verified with some fault injection experiments. In the paper we also compare our approach with other techniques from the literature.

Section 2 describes various strategies of testing processors including FPGA problems. Section 3 presents test schemes based on bijective instruction sequences and possible optimizations. Final conclusions are given in Section 4.

2. Microprocessor testing schemes

There are many papers on testing microprocessors. In general, they deal with hardware, software and mixed approaches. Hardware testing uses various built in test techniques improving test controllability and observability (they need external test controller) or even performing self-tests. These techniques are built around test path structures (e.g. scan path, circular scan path, boundary scan approach [7]). This approach involves some circuit and performance overhead. Software based testing uses specially developed test processor instruction sequences which apply test vectors to all processor resources and check their responses. This approach gains large interest in recent publications. Here we can distinguish 3 techniques: functional, structural, random and application oriented testing. A good survey of these approaches is given in [8].

Functional tests check the correctness of all instructions taking into account appropriate arguments which sensitizes various functional blocks. Taking into account the fact that a single instruction may use more than one functional block (e.g. instruction decoder, addressing circuitry, ALU, registers) we can assume orthogonality of these resources and check them independently to reduce the number of test cases. Checking instruction execution we have to fix the states of the environment (e.g. registers) and verify the results taking into account the fact that some micro operations can be skipped or falsely added. Such tests can be further extended by checking dependencies between instructions. Hence, assuring high fault coverage in functional tests is not simple and results in long tests [8].

Structural testing basses on deep knowledge of the logical structure of the microprocessor and test cases are generated in accordance to the assumed fault models within these structures. Inclusion of sophisticated fault models (e.g. delays) leads also here to test complexity. Functional and structural tests are targeted at specific fault models, usually simplified and not complete. This problem in a large extent can be resolved with pseudorandom tests.

The basic idea of pseudorandom tests is to generate subsequent instruction codes and arguments in a pseudorandom way (using

appropriate generators) and check periodically the states of registers as well as selected memory cells. The generation process takes into account some restrictions related to valid opcodes, arguments, etc. Periodical checking may base on calculating signatures over the used storage resources. In this way we can generate long test sequences in an automatic way. The longer the sequence, the better fault coverage (including unknown fault models).

In the case of realizing a fixed application (e.g. a program in an embedded system) we can be interested in testing only the correct execution of this program, not the correct operation of the whole microprocessor (some resources can be not used in this program, moreover instruction sequences are fixed in a large extent). Hence, developing application based tests we have to find representative input data to assure a representative coverage of the program control flow graph and used functional blocks. In practice, it is reasonable to combine different approaches to testing.

In the case of microprocessors configured within FPGAs we can try to use the presented schemes of testing developed for fixed logic microprocessors. However, we face two problems: usually the available resources (e.g. program memory) are limited, the embedded microprocessor structure is assured by the appropriate state of the configuration memory. Hence, the models of possible faults are more complex. Here, we have to take into account the problem of faults within the configuration memory as well as in configuration control circuitry (e.g. multiplexers, connection switches). In the literature there are several papers on testing complete FPGAs including functional blocks and configuration capabilities, e.g. [2, 4, 5, 6] and references therein. Most of them base on configuring BIST circuitry and reconfiguration processes. This leads to time consuming tests and the need of some external testing environment. Checking reconfiguration capabilities is not needed in fixed applications and in the case of applications based on embedded microprocessor cores. We deal with this letter case.

FPGA systems based on processor cores use FPGA resources in a limited way. We map the logical structure of the processor by appropriate configuration and this configuration will not be changed (fixed). Hence, it is sufficient to test the configured structure. This can be considered as some form of application testing, however on this processor we can execute a fixed program or admit its changes. In the first case we have the higher level of a fixed application. In the sequel we discuss self-testing based on the idea of the so called bijective instruction sequences. These sequences assure that an output of one sequence (test result) is used as an input (test pattern vector) for the subsequent one. This idea has been proposed by the first author in the previous papers [9,10]. In the sequel, we concentrate on extending and optimizing this approach. It will be illustrated for 8 bit microprocessor PicoBlaze [11].

3. Optimizing tests for CPU cores in FPGAs

When developing a test program for the processor we tried to reduce memory resources to store test patterns and their responses. The basic idea is using bijective instruction sequences (Section 2). The structure of such a sequence is outlined below:

1. Take an initial test pattern or a result from the previous sequence
2. Create a sequence of tested instructions with bijective property such that the final result differs from the initial pattern in the sequence
3. Repeat n times steps 1-2 and then go to the next bijective sequence.

In the simplest case we may have only one instruction in step 2 if its output differs from the initial pattern. Creating multi instruction sequences is more efficient, however it may happen that to assure bijective property these instructions should be supplemented with some modification instructions. The test is a composition of bijective sequences which cover all processor instructions. To assure high fault coverage, the appropriate number of iterations (n) should be performed. In the case of 8 bit processors we can try to create maximal iteration cycles. It is not easy to find such cycles, some previous studies resulted in up to n=40 cycles, for some instructions [9, 10]. The most critical was finding such cycles for shift instructions. We have resolved this problem using the concept of Linear Feedback Shift Register (LFSR).

LFSR is a shift register in which inputs to flip-flops are linear functions of the previous states of selected flip-flops. This feedback is defined by the so called characteristic polynomials. To achieve the maximal period of the generated states, the appropriate polynomials have to be selected. They are available in the literature, e.g. [7]. For the illustration in Fig. 1 we give LFSR based on polynomial (operator + denotes XOR function): $x^8 + x^6 + x^5 + x^4 + 1$.

The LFSR structure is created by taking the XOR function of the 8^{th}, 6^{th}, 5^{th} and 4^{th} (non-zero coefficients of x variable) flip-flop and applying it to the left most flip-flop input. This is the so called Fibonacci architecture. Fig. 1 shows the LFSR structure with shift left function, we can create a symmetrical structure with shift right. Anther structure (Galois architecture) is possible with a feedback applied from the 8^{th} flip-flop to the inputs of the appropriate flip-flops (XORed with the outputs of selected flip-flops).

The initial value of the LFSR is called the seed. In the case of non-zero seed the LFSR generates (pseudrandomely) a cycle of $2^n - 1$ (number of flip-flops), in our case 255. For the seed number 0 the LFSR does not change its state. The LFSR can be implemented in hardware or software,. We use the software implementation which is embedded into a bijective sequence of instructions.

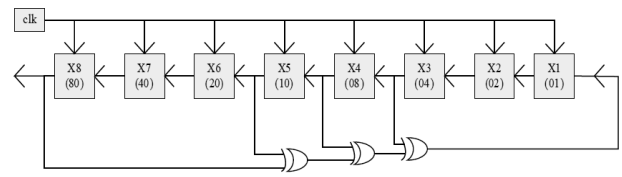


Fig. 1. LFSR structure with shift left direction (Fibonacci structure)
Rys. 1. Struktura LFSR dla przesuwania w lewo (struktura Finonacci)

```
LOAD    sC, s0  ; initial test vector in register sC
Loop1:          ;
TEST    sC, 80  ; TEST and ADDC access appropriate bit in
ADDC    s9, 00  ; the register. In this way inputs
LOAD    s8, s9  ; are prepared for „XOR" LOAD
LOAD    s9, 00  ; the subsequent instructions emulate
TEST    sC, 20  ; LFSR circuit of Figure 1
ADDC    s6, 00
XOR     s8, s6
LOAD    s6, 00
TEST    sC, 10
ADDC    s5, 00
XOR     s8, s5
LOAD    s5, 00
TEST    sC, 08
ADDC    s4, 00
XOR     s8, s4
LOAD    s4, 00
TEST    s8, 01  ;
ADDC    s7, 00  ; the last instruction related to LFSR
SL1     sC      ;   executing and testing SL1
AND     sC, FE
OR      sC, s7
LOAD    s7, 00
.
adequate cyclic block for
SL0 instruction testing
.
adequate cyclic block for
SLA instruction testing
.
adequate cyclic block for
SLX instruction testing
.
JUMP loop1 (repeated n times)
unique result in register sC (input register for the next block)
……
```

Fig. 2. Bijective test sequence for shift left instructions
Rys. 2. Sekwencja testowania bijektywnego dla instrukcji przesuwania w lewo

The bijective instruction sequences for testing all shift left instructions is given in Fig. 2. In this block we test 4 instructions: SL0, SL1, SLA, SLX. Shift instructions SL1, SR1 provide value „1'' to 0^{th} bit (SL1) or to 7^{th} bit (SR1). Similarly instructions SL0, SR0 introduce values „0" on proper bits, and instructions SLX, SRX hold LSB or MSB states of the target register, respectively.

Other bits are updated with the values from the neighboring positions (shift operation).

The sequence presented in Fig. 2 starts with an initial test pattern loaded to sC register from s0 register. Then, on the basis of bits which have influence on the next input, the so called taps of the LFSR, a new input value of the target shifted register is calculated. To access these „sensitive bits" inside the shifted register, TEST and ADDC instructions are executed upon the target register. XOR instructions are utilized to calculate the next input bit to the register. So, before execution of the target shift instruction, a new input value is already calculated. This value is placed into the register with the aid of auxiliary instructions such as AND, OR, after the shift instruction execution.

In a similar way we construct bijective test sequences for shift right (SR0, SR1, SRX, SRA), rotate left (RL) and right (RR) instructions. It is worth noting that arithmetical shift instructions as Shift Arithmetical Right (SRA), Shift Arithmetical Left (SLA) possess the ability to introduce a new input bit calculated by the LFSR mechanism without auxiliary instructions. It is ensured by the possibility of loading carry flag, adequately to bit 7^{th} (SRA), or to bit 0^{th} (SLA).

Each of the developed bijective blocks is tested in a loop, where the output result is used as a new input test pattern. Such a loop is repeated n (maximal n=254) times, and then an output result from the given block is obtained. This result serves as a new input for a next bijective block of instructions and so on. Testing with input pattern equal to „0'' (not available in LFSR) can be performed at the end, after all iterations in the loops.

Bijective blocks for other instructions need individual manipulations or calculations [9]. In the case of AND, OR, STORE-FETCH, LOAD instructions, testing the output results can be equal to the input arguments so the full cycle is easily achieved for the generated arguments (e.g. by incrementing). For XOR instruction testing some additional manipulation is needed. Instructions such as SUB,ADD, SUBCY, ADDCY, COMPARE, TEST can be tested in their bijective blocks as in the example:

```
Loop:                    ; (x255)
……..
        Bijective block to previous
         Instruction testing
……
        OR      sE, 00        ; OR test
        OR      sE, sE
        LOAD    s7, 00
        OR      s7, sE
……
         ADD    s7,  01     ;  ADD test
         JUMP   NC, loop
         ADDC   s7,  00
         SUB    s7,  01
         JUMP   loop
         unique result for every unique
          primary input test
```

Now the question arises how to extend this approach for processors with wider words (e.g. 32 bits). Here, we can perform shorter loops, e.g. partitioning the words in k bit segments (e.g. k = 8) and then repeat them with shifted segments.

We also analyzed the possibility of testing sequences with fixed input test vectors. In this case fault coverage will be reduced, nevertheless some optimization can be performed. For this purpose we used fault injection simulating SEU effects. The developed fault injection scheme was targeted at faults in the configuration memory. For a specified relatively large representative fault set we used 3 algorithms: A1 – selection of initial test patterns covering large number of faults [10], A2 – selection of test pattern detecting the most difficult faults, A3 – cyclic test with a limited number of iterations. The most difficult faults are such that are detected by a small number of test vectors e.g. 1, 2 or 3. Basing on the algorithm A1 we obtained 84% fault coverage with 4 test vectors, for 18 vectors this increased to 85.6%, so the effect of saturation was visible. It is worth noting that a single test vector assured 69.5% coverage. With use of the algorithm A2 we got better results, 79% fault coverage for one of the vectors sensitizing the most difficult fault (with a unique test pattern). This test vector detected beyond this fault many other faults. Practically, for 28 vectors the fault coverage exceeded 99%, for 3 and 10 vectors we obtained 90% and 98%, respectively. In the case of the algorithm A3 we obtained 90% coverage for n=20. For more complex processors the effectiveness of tests based on a small number of test vectors will not be so efficient, hence the algorithm A3 is a good solution here. Moreover, here we consider only single bit configuration faults. However, this shows significant capabilities of optimization.

Further optimization can be performed by limiting testing the processor only to a specified application program. In this case not all processor resources (including the configuration memory) are used, so there is a bigger potential for reducing the number of test cases. Here, we have only to find the representative coverage of input data to cover the program flow graph. For some simple programs e.g. matrix multiplication we got successful results for a single or a few program execution.

4. Conclusions

In the paper we have shown that typical FPGA applications designed on the basis of embedded processor cores create some possibility of simplifying tests covering not only classical fault models typical for microprocessors but also these related to the configuration circuitry. The efficient test procedures are also the basis for designing fault tolerant systems based on FPGAs [1, 4]. The effectiveness of the tests has been verified only for SEUs in the configuration memory. We plan to check also their effectiveness for other faults. This will result in more test vectors related to calculation units (e.g. adder, multiplier), however this is a relatively simple task (compare [7]). Another issue is checking the configuration memory via read back operations (we did this in [6]). Unfortunately, this leads to high time overhead, moreover we have to check all configuration bits despite the fact that only a small fraction of them has the impact on the embedded processor or the realized program.

5. References

[1] Bolchini C., Miele A., Sandionigi C.: TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGA, IEEE Int. Symposium on Defect and Fault Tolerance in VLSI Systems, 2007.

[2] Dutton B., Stroud C.: Soft core embedded processor based built-in self-test of FPGAs, 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2009.

[3] Lesea A., et al.: The rosetta experiment: atmospheric soft error rate testing in differing technology FPGAs, IEEE Transactions on Materials Reliability, September, 2005.

[4] Legat U., Biasizzo A., Nowak F.: On-line self-recovery of embedded multi-processor SOC on FPGA using dynamic partial reconfiguration, Information Technology and Control, Vol. 41, No. 2, 2012.

[5] Renovell M, et al.:, Testing the interconnect of RAM based FPGAs, IEEE Design and Test of Computers, Vol. 15, No.1 1998.

[6] Sosnowski J., Pawłowski M.: Improving Testability in systems with FPGAs, 22nd Euromicro Conference, Short Contributions (ed. B. Werner), IEEE Computer Society Press, ISBN 0-8186-7703-1, 1996.

[7] Sosnowski J.: Testowanie i niezawodność systemów komputerowych, Exit 2006.

[8] Sosnowski J.: Software based self-testing of microprocessors, Journal of Systems Architecture 52 (2006) pp.257-271.

[9] Wegrzyn M., Biasizzo A., Novak F.: Application-oriented testing of embedded processor cores implemented in FPGA circuits, International Review on Computers and Software, 2007, Vol. 2, No. 6.

[10] Wegrzyn M., Novak F, Biasizzo A.: functional testing of processor codesin fpga-based applications", Computing and Informatics, Vol. 28, 2009.

[11] PicoBlaze 8-bit Embedded Microcontroller, User Guide for Spartan-3, Virtex-II, andVirtex-II Pro FPGAs, www.xilinx.com, 1-800-255-7778 UG129 (v1.1.1) November 21, 2005, UG129 (v2.0) June 22, 2011.

otrzymano / received: 09.04.2014
przyjęto do druku / accepted: 02.06.2014 *artykuł recenzowany / revised paper*