Parity Codes Used for On-line Testing in FPGA

Pavel Kubalík, Hana Kubátová

This paper deals with on-line error detection in digital circuits implemented in FPGAs. Error detection codes have been used to ensure the self-checking property. The adopted fault model is discussed. A fault in a given combinational circuit must be detected and signalized at the time of its appearance and before further distribution of errors. Hence safe operation of the designed system is guaranteed. The check bits generator and the checker were added to the original combinational circuit to detect an error during normal circuit operation. This concurrent error detection ensures the Totally Self-Checking property. Combinational circuit benchmarks have been used in this work in order to compute the quality of the proposed codes. The description of te benchmarks is based on equations and tables. All of our experimental results are obtained by XILINX FPGA implementation EDA tools. A possible TSC structure consisting of several TSC blocks is presented.

Keywords : *On-line testing, self-checking, error detection code, fault, error, FPGA*

1. Introduction

The design process for FPGAs differs mainly in the “design time”, i.e., in the time needed from the idea to its realization, in comparison with the design process for ASICs. Moreover, FPGAs enable different design properties, e.g., in-system reconfiguration to correct functional bugs or update the firmware to implement new standards. Due to this fact and due to the growing complexity of FPGAs, these circuits can also be used in mission-critical applications such as aviation, medicine or space missions.

There have been many papers [1, 2] on concurrent error detection (CED) techniques. CED techniques can be divided into three basic groups according to the type of redundancy. The first group focuses on area redundancy, the second group on time redundancy and the third one on information redundancy. When we speak about area redundancy, we assume duplication or triplication of the original circuit. Time redundancy is based on repetition of some computation. Information redundancy is based on error detecting (ED) codes, and leads either to area redundancy or time redundancy. Next, we will assume the utilization of information redundancy (area redundancy) caused by using ED codes.

The process when high-energy particles impact sensitive parts is described as a Single Event Upset (SEUs) [3]. SEUs can lead to bit-flips in SRAM. The FGPA configuration is stored in SRAM, and any changes of this memory may lead to a malfunction of the implemented circuit. Some results of SEU effects on FPGA configuration memory are described in [4]. CED techniques can allow faster detection of a soft error (an error which can be corrected by a reconfiguration process) caused by an SEU. SEUs can also change values in the embedded memory used in the design, and can cause data corruption. These changes are not detectable by off-line tests, only by some CED techniques. The FPGA fabrication process allows the use of sub-micron technology with smaller and smaller transistor size. Due to this fact the changes in FPGA memory contents, affected by SEUs, can be observable even at sea level. This is another reason why CED techniques are important.

There are three basic terms in the field of CED:

- The Fault Security (FS) property means that for each modeled fault, the produced erroneous output vector does not belong to the proper output code word.
- The Self-Testing property (ST) means that, for each modeled fault, there is an input vector occurring

during normal operation that produces an output vector which does not belong to the proper output code word.

- The Totally Self-Checking (TSC) property means that the circuit must satisfy FS and ST properties.

The basic method for the proper choice of a CED model is described in [5]. Techniques using ED codes have also been studied by other research groups [6, 7]. One method is based on a parity bits predictor and a checker, see Figure 1.

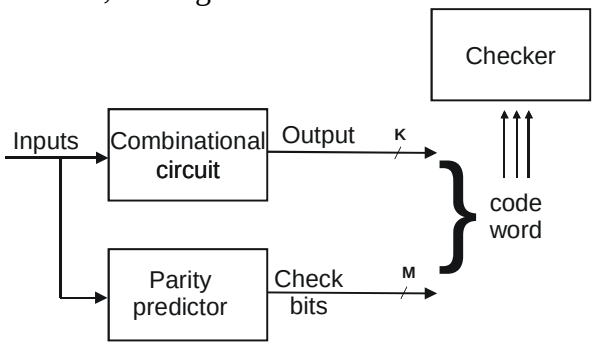

Fig. 1: Structure of a TSC circuit

2. The fault model

All of our experiments are based on FPGA circuits. The circuit implemented in an FPGA consists of individual memory elements (LUTs - look up tables). We can see 3 gates mapped into an LUT in Figure 2.

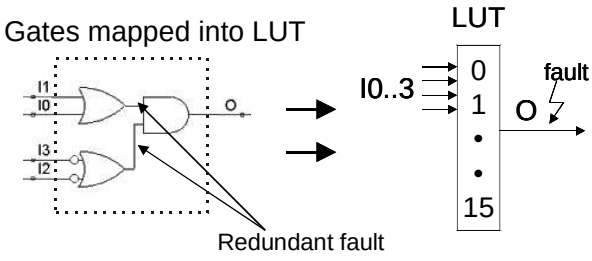

Figure 2. Fault model

The original circuit has two inner nets. The original set of the test vectors covers all faults in these inner nets. These test vectors are redundant for an LUT. For circuits realized by LUTs a change (a defect) in the memory leads to a single event upset (SEU) at the primary output of the LUT. Therefore we can use the stuck-at fault model in our experiments to detect SEU – only some of the detected faults will be redundant.

Our fault model is described by a simple example in Figure 3. Only one LUT is used for simplicity. This LUT implements a circuit containing 3 gates. The primary inputs from I0 to I1 are the same as the address inputs for the LUT. When this address is selected its content is propagated to the output.

We assume the following situation: first the content of this LUT can be changed, e.g., electromagnetic interference, cross-talk or alpha particles. The appropriate memory cell is set to one and the wrong value is propagated to the output. This means that the realized function is changed and the output behaves as a single event upset. We can say that a change of any LUT cell leads to a stuck-at fault on the output according to this example. This fault is observed only if the bad cell is selected. This is the same situation as for circuits implemented by gates. Some faults can be masked and do not necessarily lead to an erroneous output.

Due to masking of some faults, the possibility of their appearance can occur at the time when previously unused logic is being used. E.g., if one bit of an LUT is changed, the erroneous output will appear, while the appropriate bit in an LUT is selected by the address decoder.

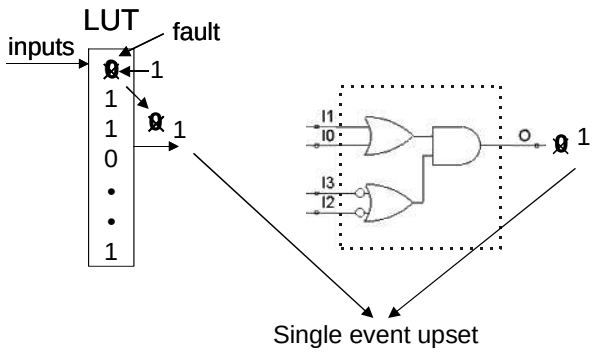

Figure 3. Fault Model – Example

In our design methodology we evaluate FS and ST properties. For ST properties a hidden fault is not assumed.

The evaluation of the FS property is independent of the set of allowed input words. If a fault does not manifest itself as an incorrect codeword for all possible input words, it cannot cause an undetectable error for any subset of input words. So we can use the exhaustive test set for combinational circuits.

The exhaustive test set is generated to evaluate the ST property for combinational circuits, where the set of input words is not defined. But in a real situation, some input

words may not occur. This means that some faults can be undetectable. This can decrease the final fault coverage. Therefore, the number of faults that can be undetectable is higher.

The fault simulation process is performed for circuits described by netlist (for example .edif).

3. Parity bits predictor

There are many ways to generate checking bits. A single even parity code is the simplest code that may be used to get a code word at the output of the combinational circuit. This parity generator performs XOR over all primary outputs. However, the single even parity code is mostly not appropriate to ensure the TSC goal.

Another error code is a Hamming-like code, which is in essence based on the single parity code (multi parity code). The Hamming code is defined by its generating matrix. We used a matrix containing the unity sub-matrix on the left side for simplicity. The generating matrix of the Hamming code (15, 11) is shown in Fig. 4. The values a_{ij} have to be defined.

When a more complex Hamming code is used, more values have to be defined. The number of outputs o_i used for the checking bits determines the appropriate code. E.g., the circuit alu1 [10] having 8 outputs requires at least the Hamming code (15, 11). Therefore 8 data bits and 4 checking bits are used. The definition of the values a_{ik} is also important.

Now we present a method for generating values a_{ik}. Let us mention the Hamming code (15, 11) having 4 checking bits. In our case (alu1) we have only 8 bits. Therefore the reduced Hamming matrix must be used.

$$G = \left(\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ 0 & 1 & \cdots & 0 & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & a_{11,1} & a_{11,2} & a_{11,3} & a_{11,4} \end{array}\right)$$

Fig. 4. Generating matrix for Hamming code (15, 11)

The sub-matrix has only 8 rows and 4 columns after the reduction. We can define eight 4-bit vectors or four 8 bit vectors. The second case will be used here. The search for erroneous output is a similar method to a binary search. The first vector is composed of log. 1s only. The last vector is composed of log. 1s in the odd places and log. 0s in the even places. Every vector except the first contains the same number of 1s and the same number of 0s. An example of the possible content of the right part sub-matrix is shown in Fig. 5.

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig. 5. Right part of generating matrix

The number of vectors in the set is the same as the number of rows in the appropriate Hamming matrix. The way to generate parity output for checking bit x_k is described by equation 1:

$$x_k = a_{1k}o_1 \oplus a_{2k}\, o_2 \oplus \ldots \oplus a_{mk}o_m, \qquad (1)$$

where $o_1 \ldots o_m$ are the primary outputs of the original circuit.

4. Area overhead minimization

The benchmarks used in this paper are described by a two-level network. The final area overhead depends on the minimization process. We used two different methods in our approach. Both these methods are based on a simple duplication of the original circuit.

Our first method is based on a modification of the circuit described by a two-level network. The area of the check bits generator contributes significantly to the total area of the TSC circuit. As an example we consider a circuit with 3 inputs (c, b and a) and 2 outputs (f and e). The check bits generator uses the odd parity code to generate the check bits. In our example we have only one check bit x.

Our example is shown in Table 1. Output x was calculated from outputs e and f. We have to generate the minimal form of the equation at this time. We can achieve the minimal form using methods like the

Karnaugh map or Quine-McCluskey. After minimization we obtain three equations, one per output (f, e and x), where x means an odd parity of the outputs f and e. If we want to know whether the odd parity covers all faults in our simple combinational circuit example, we have to generate the minimal test set and simulate all faults in each net in this circuit.

Table. 1. Example of parity generator

c b a	f e	x
0 0 0	0 1	0
0 0 1	1 0	0
0 1 0	1 0	0
0 1 1	1 0	0
1 0 0	0 1	0
1 0 1	0 1	0
1 1 0	1 1	1
1 1 1	0 0	1

The final equations are:

$$e = bc + a(b + c) \quad (2)$$
$$f = ab + c(a + b) \quad (3)$$
$$x = bc \quad (4)$$

Our second method is based on a modification of the multi-level network. The parity bits are incorporated into the tested circuit as a tree composed of XOR gates. The maximal area of the parity generator can be calculated as the sum of the original circuit and the size of the XOR tree.

5. Experimental Evaluation Software

Fig. 6. describes how the test is performed for each detecting code. The MCNC benchmarks [11] were used in our experiments. These benchmarks are described by a truth table. To generate the output parity bits, all the output values have to be defined for each particular input vector. Only several output values are specified for each multi-dimensional input vector, and the rest are assigned as don't cares; they are left to be specified by another term. Thus, in order to be able to compute the parity bits, we have to split the intersecting terms, so that all the terms in the truth table are disjoint.

In the next step, the original primary outputs are replaced by parity bits. Two different error codes were used to calculate the output parity bits (single even parity code and Hamming code). Another tool was used in the case where the original circuit was modified in multilevel logic. This tool is described in [8]. Two circuits generated in the first step (the original circuit and the parity circuit) are processed separately to avoid sharing any part of the circuit. Each part is minimized by the Espresso tool [9]. The final area overhead depends on the software that was used in this step. Many tools were used to achieve a small area of the parity bits generator. Only Espresso was used to minimize the final area of the circuit described by the two level network. In this step the area overhead is known for implementation to ASIC. For FPGAs the area overhead is known after the synthesize process has been performed.

The "pla" format is converted into the "bench" format in the next step. The "bench" format was used because, the tool which generates the exhaustive test set uses this format. An exhaustive test set has 2^n patterns, and we used it to evaluate the TSC goals.

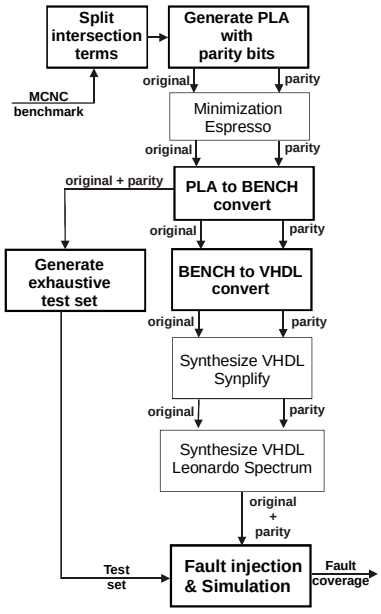

Fig. 6. Design scheduling of self-checking circuit

Another conversion tool is used to generate two VHDL codes and the top level. The top level is used for incorporating original and parity circuit generator. In the next step, the synthesis process is performed by Synplify [12]. The constraints properties set during the synthesis process express the area overhead and the fault coverage. If the maximum frequency is set too high, the synthesize process causes hidden faults to occur during the fault simulation. The hidden

faults are caused by circuit duplication or by the constant distribution. The size of the area overhead is obtained from the synthesis process. The final netlist is generated by the Leonardo Spectrum [13] software. The fault coverage was obtained by simulation using our software.

6. Software solution description

Special tools had to be developed to evaluate the area overhead and fault coverage. In addition to some commercial tools such as Leonardo Spectrum [13] and Synplify [12] we used format converting tools, parity circuit generator tools and simulation tools.

At first, area minimization and term splitting is performed for the original circuit by BOOM [10]. The Hamming code generator (or single parity generator) is generated by the second software. These two circuits are minimized again with Espresso. The next two tools convert the two-level format into a multi level format. The first converts a “pla” file to “bench”, and the second converts “bench” to VHDL. The second software is used for generating the final circuit in the “bench” format for further usage in the exhaustive test set generator. The format converting software and parity generator software were written in Microsoft Visual C++. The netlist fault simulator was written in Java. The parser source code was used for parsing the netlist that is generated by the two commercial tools described above.

7. Experiments

The combinational MCNC benchmarks [11] were used for all the experiments. These benchmarks are based on real circuits used in large designs.

Since the whole circuit will be used for reconfiguration in FPGA, only small circuits were used. Real designs having a large structure must by partitioned into several smaller parts. For large circuits, the process of area minimization and fault simulation takes a long time. This disadvantage prevents us examining more methods of designing the check bits generator.

The evaluated area, FS and ST properties depend on circuit properties such as the number of inputs and outputs, and the circuit complexity. The experimental results show that a more important property is the structure of the circuit. Two basic properties are described in Table 2.

Table 2 Description of tested benchmarks

Circuit	Inputs	Outputs
alu1	**12**	**8**
apla	**10**	**12**
b11	**8**	**31**
br1	**12**	**8**
al2	**16**	**47**
alu2	**10**	**8**
alu3	**10**	**8**
c17	**5**	**2**

In the first set of experiments our goal was to obtain one hundred percent of the FS and ST property, while we measured the area overhead. In this case, the maximum of the parity bits was used.

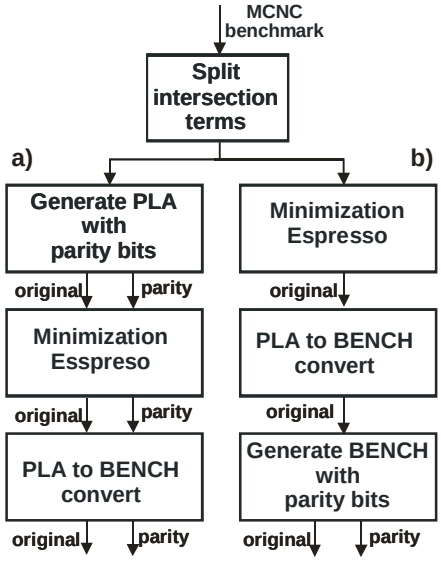

Fig. 7. Two different flows for creating a parity generator

This task was divided into two experiments (Fig. 7). In the first experiment the two-level network was being modified (Fig. 7a). The results are shown in Table 3.

Table 3 Hamming code – PLA

Circuit	Parity nets	Original [LUT]	Parity [LUT]	Overhead [%]	ST	FS
alu1	**4**	**8**	**84**	**1050**	**100**	**100**
apla	**5**	**45**	**105**	**233**	**100**	**98,3**
b11	**6**	**38**	**38**	**100**	**100**	**99,7**
br1	**4**	**50**	**59**	**118**	**100**	**95,9**
al2	**7**	**51**	**54**	**106**	**100**	**98,8**
alu2	**4**	**30**	**127**	**423**	**100**	**100**

alu3	**4**	**28**	**94**	**336**	**100**	**100**
c17	**2**	**2**	**3**	**150**	**100**	**100**

The ST property was fulfilled in 7 cases and the FS property was fulfilled in 4 cases. The area overhead in many cases exceeds 100%. This means that the cost of one hundred percent fault coverage is too high. In these cases the TSC goal is satisfied for most tested benchmarks.

We then used an old method, where the original circuit described by a multi-level network is modified by additional XOR logic (Fig. 7b) [8].

Table 4 Hamming code – XOR

Circuit	Parity nets	Original [LUT]	Parity [LUT]	Overhead [%]	ST	FS
alu1	**4**	**8**	**13**	**163**	**100**	**100**
apla	**5**	**45**	**114**	**253**	**100**	**97,2**
b11	**6**	**38**	**73**	**192**	**100**	**99**
br1	**4**	**50**	**85**	**170**	**100**	**96,5**
al2	**7**	**52**	**109**	**210**	**100**	**99,1**
alu2	**4**	**30**	**52**	**173**	**100**	**100**
alu3	**4**	**28**	**44**	**157**	**100**	**100**
c17	**2**	**2**	**3**	**150**	**100**	**100**

The results obtained from this experiment are shown in Table 4. The FS and the ST properties were fulfilled in the same cases as in the first experiment, but the overhead is in some cases smaller.

In the second set of experiments we tried to obtain a small area overhead, and the fault coverage was measured. In this case the minimum of parity bits is used (single even parity).The experiments are divided into two groups, a) and b), Fig. 7. The procedure is the same as described above.

In the first experiment the two-level network of the original circuit was modified (Fig. 7a). The results are shown in Table 5.

Table 5. Single even parity – PLA

Circuit	Parity nets	Original [LUT]	Parity [LUT]	Overhead [%]	ST	FS
alu1	**1**	**8**	**271**	**3388**	**100**	**98,9**
apla	**1**	**46**	**23**	**50**	**99,5**	**82,6**
b11	**1**	**37**	**3**	**8**	**89,9**	**77,3**
br1	**1**	**54**	**10**	**19**	**86,9**	**62,1**
al2	**1**	**52**	**4**	**8**	**97,3**	**91,7**
alu2	**1**	**29**	**47**	**162**	**100**	**91,2**
alu3	**1**	**26**	**32**	**123**	**100**	**92**
c17	**1**	**2**	**2**	**100**	**100**	**100**

The ST property is achieved in four cases, but the area overhead is smaller in five cases. The FS property is satisfied in one case.

In the last experiment, we have modified the circuit described by a multilevel network (Fig. 7b). The ST property was satisfied in four cases and the FS property in two cases. The area overhead is higher than 100% for most benchmarks, but the fault coverage did not increase, Table 6.

Table 6. Single even parity – XOR

Circuit	Parity nets	Original [LUT]	Parity [LUT]	Overhead [%]	ST	FS
alu1	**1**	**8**	**10**	**125**	**100**	**100**
apla	**1**	**46**	**56**	**122**	**99,7**	**87,2**
b11	**1**	**37**	**36**	**97**	**93,9**	**81,4**
br1	**1**	**54**	**61**	**113**	**92,7**	**69**
al2	**1**	**52**	**23**	**44**	**97,9**	**93,2**
alu2	**1**	**29**	**44**	**152**	**100**	**91,1**
alu3	**1**	**26**	**39**	**150**	**100**	**91,6**
c17	**1**	**2**	**2**	**100**	**100**	**100**

8. Huge design

Our previous results show that it is in many cases too difficult to achieve TSC goals with minimal area overhead [8]. A way to detect and localize the fault part of the circuit has to be proposed. Assuming that the TSC goals cannot be higher than 90%, the area overhead can be rapidly decreased, and other methods to cover and localize the fault can be used. On-line testing methods can only detect faults. The localization process must exploit some other methods for off-line testing. However, neither on-line nor off-line tests increase the reliability parameters. The reliability mostly decreases due to the larger area occupied by the TSC circuit than by the original circuit.

Therefore we propose a reconfigurable system to increase these parameters. Each block in our design is designed as a TSC, and we have been working on a methodology to satisfy TSC goals for the whole design and to design highly reliable systems. The way to connect all TSC blocks is shown in Figure 8. The main idea is based on detection of the error code word generated in any block. The detecting process is moved from the primary outputs to the primary inputs of the following circuit. The interconnections of all individual

blocks play an important role with respect to the TSC property of the whole circuit. A bad order of the connections between the inner blocks leads to lower fault coverage. Additional logic has to be included into the control arrangement of the implemented blocks with respect to the way the automatic tools handle the interconnection.

In our structure we can assume six places where an error can be observable. We assume, for simplicity that an error that occurred in the check bit generator will be observable at the parity nets (number 1) and error occurred in the original circuit will be observable at the primary outputs (number 5).

The checker in block N will detect the error if it occurs in net number 1, 2, 4 or 5. If the error occurs in the net number 3 or 6, the error will be detected in the next checker (N+1).

All our experiments were applied to combinational circuits only. The same techniques can be used for a sequential circuit, because these circuits can be divided into simple combinational parts separated by flip-flops. The finite state machine can be divided into two parts: the first part covers the combinational logic from inputs to flip-flops (with feedback), while the second part covers the combinational logic from flip-flops to outputs (and the parts connected directly from the input to the output).

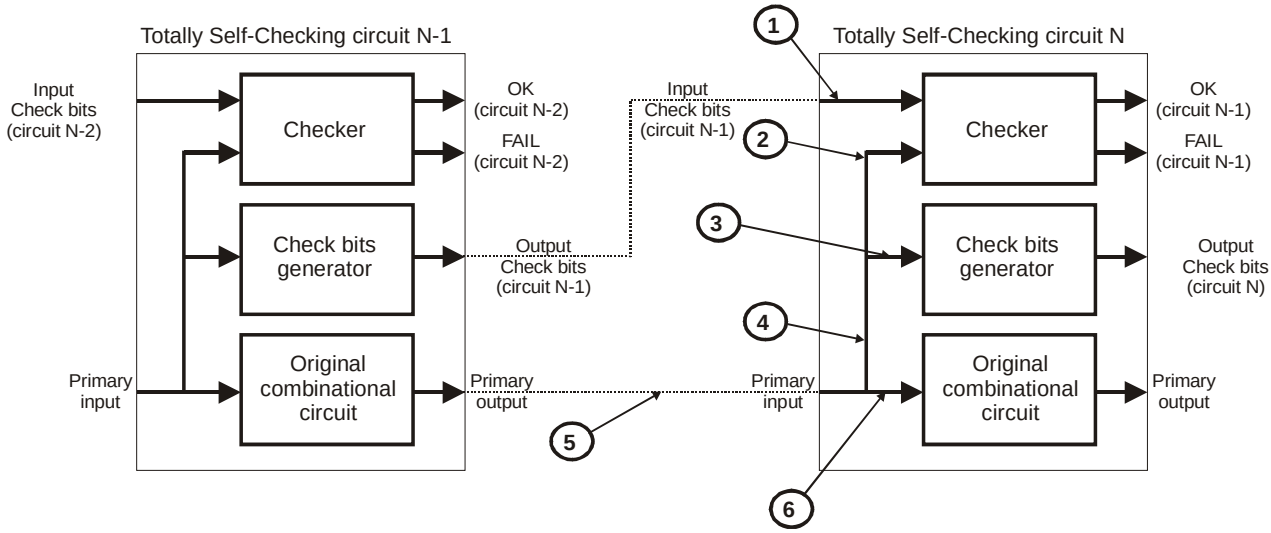

Fig. 8 Proposed structure of TSC circuits implemented in FPGA

9. Conclusion

The paper describes one part of the automatic design process methodology for a dynamic reconfiguration system. We designed concurrent error detection (CED) circuits based on FPGAs with a possible dynamic reconfiguration of the faulty part. The reliability characteristics can be increased by reconfiguration after the error detection. The most important criterion is the speed of the fault detection and the safety of the whole circuit with respect to the surrounding environment.

In summary, FS and ST properties can be satisfied for the whole design, including the checking parts. This is achieved by using more redundancy outputs generated by the special codes.

A Hamming-like code can be used as a suitable code to generate check bits. The type depends on the number of outputs and on the complexity of the original circuit [9].

More complex circuits need more check bits. We would like to reduce the duplicated circuit and compute the fault coverage again. We have proposed a new solution of the check bits generator design method. Because we want to increase the reliability characteristics of the circuit implemented in FPGAs, we have to modify the circuits at the netlist level.

All of our experiments apply combinational circuits only. Sequential circuits can be disjoint to the simple

combinational parts separated by flip-flops. Therefore this restriction only to combinational circuits does not reduce the quality of our methods and experimental results.

Our future improvements will involve d discovering closer relations between real FPGA defects and our fault models. Minimization of the whole TSC design to obtain the lowest area overhead has been under intensive experimentation. We are also working intensively on the appropriate decomposition of the designed circuit.

References

[1] Mohanram, K., Sogomonyan, E. S., Gössel, M., Touba, N. A.: Synthesis of Low-Cost Parity-Based Partially Self-Checking Circuits, Proceedings of the 9th IEEE International On-Line Testing Symposium, 2003, pp. 35.

[2] Drineas, P., Makris, Y.: Concurrent Fault Detection in Random Combinational Logic, In: Proceedings of the IEEE International Symposium on Quality Electronic Design (ISQED), 2003, pp. 425-430.

[3] QuickLogic Corporation.: Single Event Upsets in FPGAs, 2003, www.quicklogic.com

[4] Bellato, M., Bernardi, P., Bortalato, D., Candelaro, A., Ceschia, M., Paccagnella, A., Rebaudego, M., Sonza Reorda, M., Violante, M., Zambolin, P.: Evaluating the effects of SEUs affecting the configuration memory of an SRAM-based FPGA Design Automation Event for Electronic System in Europe 2004, pp. 584-589.

[5] Mitra, S., McCluskey, E. J.: Which Concurrent Error Detection Scheme To Choose? Proc. International Test Conf., 2000, pp. 985-994.

[6] Bolchini, C., Salice, F., Sciuto, D.: Design Self-Checking FPGAs through Error Detection Codes, 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'02), 2002, pp. 60.

[7] Bolchini, C., Salice, F., Sciuto, D.,Zavaglia R.: An Integrated Design Approach for Self-Checking FPGAs, 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), 2003, pp. 443.

[8] Kubalík, P. and H. Kubátová (2003). Design of Self Checking Circuits Based on FPGA. In: Proc. of 15th International Conf. on Microelectronics, pp. 378-381. Cairo, Cairo University.

[9] Brayton, R. K. et al. (1984). Logic minimization algorithms for VLSI synthesis, Boston, MA, Kluwer Academic Publishers, 192 pp.

[10] Hlavička, J. and P. Fišer (2001): BOOM - a Heuristic Boolean Minimizer. Proc. International Conference on Computer-Aided Design ICCAD 2001, San Jose, California (USA), pp. 439-442.

[11] S. Yang.: Logic synthesis and optimization benchmarks user guide. Technical Report 3, Microelectronics Center of North Carolina, 1991

[12] http://www.synplicity.com/

[13] http://www.mentor.com/