Title: Design of Low-Cost Self-Checking Circuits

Authors: C.-F. Huang and S.-J. Wang

Affiliation: Institute of Computer Science
National Chung-Hsing University
Taichung 402, Taiwan, ROC

Corresponding Author:
Sying-Jyan Wang
Inst. of Computer Science
National Chung-Hsing University
Taichung 402, Taiwan, ROC

Phone: +886-4-2840498 Ext. 910
Fax: +886-4-2853869
E-mail: sjwang@cs.nchu.edu.tw

ABSTRACT

Self-checking circuits can detect the presence of both transient and permanent faults. A self-checking circuit consists of a functional circuit, which produced encoded output vectors, and a checker, which check the output vectors. A self-checking system consists of an interconnection of self-checking circuits. The advantage of such a system is that errors can be caught as soon as they occur; thus, data contamination is prevented. Previous works on self-checking circuit design try to restrict the synthesis procedure such that only restricted errors may appear. However, this approach may not be always feasible, and always requires significant hardware overhead. In this paper, we present a new design methodology, in which functional circuit is not modified. Compared with previous methods, our approach requires less hardware overhead, and experimental results show that the method also achieves good fault coverage.

1. Introduction

The advance in VLSI technology keeps on putting more and more devices into the same silicon area, and the trend is likely to continue. It is well known that transient faults are the predominant cause of system failures [1]-[3]. With the move towards deep-submicron technology, power supply voltage level will be further reduced and thus the noise margins become smaller. The reduced feature sizes also increase the coupling capacitance between adjacent metal lines, which makes cross-talk noise a growing source of transient faults. As a result, system susceptibility to transient faults is likely to increase.

Self-checking circuits and systems can detect the presence of both transient and permanent faults. A self-checking circuit consists of a functional circuit, which produces encoded output vectors, and a checker, which checks the vector to see if an error has occurred. The checker has the ability to give an error indication even when a fault occurs in the checker itself. The functional circuit can be either combinational or sequential. A self-checking system consists of an interconnection of self-checking circuits.

The output vectors of a self-checking functional circuit are often encoded in a code that detects unidirectional errors. The reason is that unidirectional errors are very common in VLSI [3],[4]. Such an error is said to occur when there are multiple transitions in either the $0 \rightarrow 1$ direction or the $1 \rightarrow 0$ direction, but not both. For example, in [4] it was shown that a stuck-at fault, cross-point fault, ir a short in an MOS programmable logic array (PLA) or read-only memory causes only a unidirectional error. Many codes have been proposed for detecting such errors.

Various synthesis methods have been proposed to modify the functional circuit so that only unidirectional errors occur [5]-[7], while other methods try to control the number of erroneous bits at the output by restricting fanout [8]-[10]. However, in some cases it may be either difficult or impossible to modify the functional circuits. For these circuits, some other methods are required. For example, weight-based codes [11] have been proposed for concurrent error checking. The drawback of this approach, however, is that the size of checkers can be tremendous.

In this paper, we propose a new design methodology for self-checking circuits. The goal is to achieve a high level of fault coverage while at the same time reduce the hardware overhead. This paper is organized as follows. In the next section, we give some background information, while in Section 3 we present our self-checking design methodology. The experimental results are shown in Section 4 and some concluding remarks are given in Section 5.

2. Preliminaries

In this section, we provide some preliminary information.

2.1. Self-Checking Circuits

The general structure of self-checking circuits is illustrated in Fig. 1. The functional circuit carries out the normal operations and produces an n-bit output vector. The check symbol generator (CSG) generates k-bit check symbol for any given output vector. The checker check to see if the output vector along with the check bits forms a valid output code word. The outputs of checker consist of two bits so that a single stuck-at fault will not invalidate the checker.

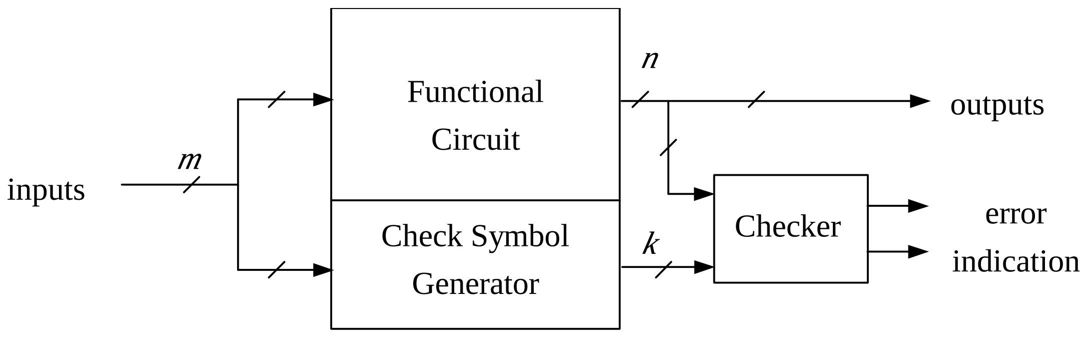

Fig. 1. Self-Checking Circuit Structure.

The most important concept developed for self-checking circuits is the totally self-checking (TSC) concepts [12],[13]. The following definitions can be used to describe a TSC system. Let F denote the set of faults, and G be the network in these definitions.

Definition 1: G is *self-testing* with respect to F if for every fault in F the circuit produces an output noncode word for at least one input code word.

Definition 2: G is *fault-secure* with respect to F if for every fault in F the circuit never produces an incorrect output code word for any input code word.

Definition 3: G is *TSC* if it is both self-testing and fault-secure.

2.2. Error Detecting Codes

A TSC checker guarantees the detection of the first error under the single fault assumption, no matter where the fault is. Many TSC checker designs can be found in the literature. However, it is usually much more difficult to ensure TSC properties in the functional circuits. Many design procedures have been proposed for self-checking circuits. For example, in [5]-[7] the functional circuit is designed in such a way that only unidirectional errors may appear. The outputs of the functional circuit and the CSG form a unidirectional error detecting (AUED) code which can be checked by the checker. A circuit designed in such a way, however, may not be TSC.

Many error-detecting codes have been used to encode the circuit output. One commonly used code is the Berger code [14], which is the optimal systematic AUED code. In a Berger code, the check symbol can be obtained by counting the number of zeros in the information bits, and hence the encoding/decoding scheme is simple. The TSC checkers for Berger code have been proposed, and they are easy to implement. Berger code encoding is useful for circuit with regular structure such as PLA's and ALU's, in which the types of errors are restricted. However, in most multi-level circuits, a single fault in the circuit may produce symmetric errors at the outputs, and these errors may not be detectable under Berger code encoding.

For most circuits, the fault coverage achieved by Berger code encoding scheme is not high enough. Weight-based codes [11] were developed to deal with such a problem. In a weigh-based code, each output bit is assigned with a weight, and the check symbol is simply the sum of all output bits multiplying their respective weights. Berger code can be seen as a special case of the weight-based code, in which all weights are 1's. The probability of *alias* (i.e., error masking) usually decreases as the number of distinct weights increases; however, the check symbol become longer and the checker will be much more complicated, and it may not be easy to find a set of good weights for a given circuit.

2.3. TSC Checkers

Many TSC checkers have been proposed for various encoding schemes. One of the checkers that we will use in the later part of this paper is the two-rail checker (TRC) [13]. Two-rail checkers satisfies the TSC properties, and they can used to compare whether two sets of equal-length data are identical. A 2-bit two-rail check

(TRC$_2$) is shown in Fig. 2, and it can be used to compare the two sets (a_1b_1) and (a_2b_2). Other two-rail checkers can be constructed by using TRC$_2$ as the building blocks.

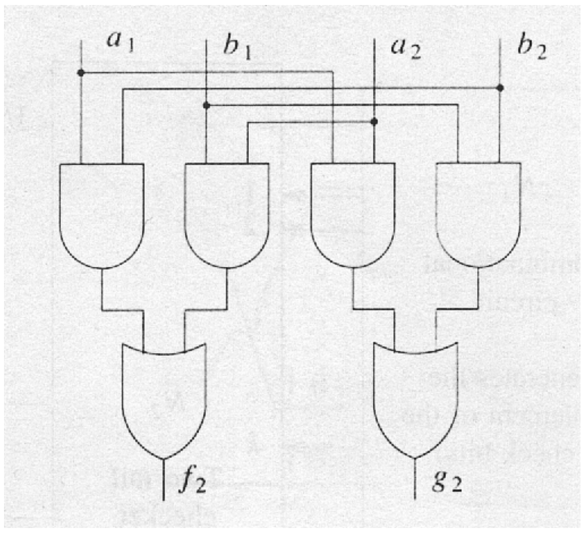

Fig. 2. A 2-bit two-rail check.

3. CED Structure

In this section, we present our concurrent error detecting structure, and give a basic analysis.

3.1. Proposed Structure

In the proposed method, we do not apply any specific error detecting code, as did in all previous methods. In contrast, we define the checker structure first, and the check symbols are generated accordingly. The goal is to design a self-checking scheme with a relative small check symbol while at the same time achieves high fault coverage. Let the length of the check symbol be k. With a smaller k, the CSG is usually smaller, which means the hardware overhead is smaller. On the other hand, a small k also increases the chance of alias, so should be dealt with by a better encoding scheme. The general self-checking circuit structure for our method is shown in Fig. 3.

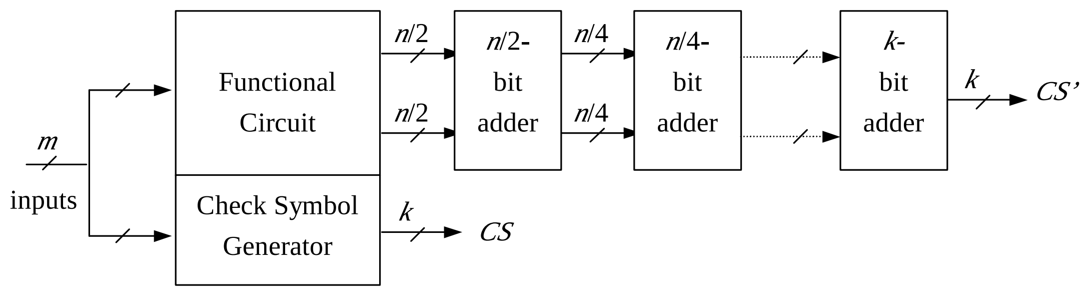

Fig. 3. Proposed Self-Checking Circuit Structure.

The *n*-bit outputs are divided into two parts, which are fed into an *n*/2-bit adder. The adder outputs are divided into two parts again, and then they are fed into the second adder the further reduce the number of outputs. This process is repeated until we have a *k*-bit check symbol *CS'*. This regenerated check symbol (*CS'*) is then compared with the check symbol generated from CSG (*CS*) by s two-rail checker.

The encoding process can be explained by the example shown in Fig. 4. Let *k* be 3. In the first example (left one), *n*=7 and the output vector is 0100110. This vector is divided into two parts and added, and the result is 01010. This process is repeated until we obtain a 3-bit check symbol, which is 001. In the second example, *n*=10 and the output vector is 1011010001, and the final check symbol is 101.

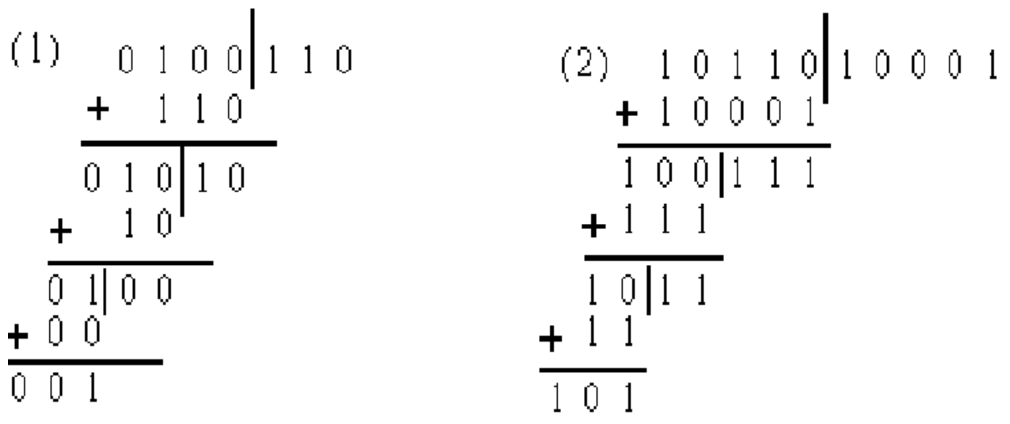

Fig. 4 An example.

3.2. Analysis

The quality of an encoding scheme is partially decided by how the information bits are mapped to the check symbols. An error will be undetectable if an *alias* occurs, so that the check symbols for faulty and fault-free output vectors are the same. For example, consider the Berger code, in which check symbol is simply the zero count of the information bits. Let n=4 and k=3. In this case, there are 16 output vectors, in which 6 vectors are mapped to the check symbol 010 (i.e., 2 in decimal). Check symbols 101, 110, and 111 are not used, since the number of 0's is at most 4 in a 4-bit vector.

In Table I we show the distribution of check symbols for n=4 and k=3 under three different encoding schemes: Berger code, Add, and Sub. In the proposed self-checking scheme, adders are used to compress output vectors into check symbols. Since an adder can conduct both addition and subtraction operations, both operations are tried in the encoding scheme. The 8 different check symbols are listed, and the number of output vectors mapped to the symbol are shown in the Table. The number given in each parenthesis is the minimum Hamming distance between any two vectors whose check symbols are identical. This is the minimum number of erroneous bits enquired to create an alias. In this Table, it can be seen that the distribution of Hamming code encoding is not uniform and many check symbols are not used. Therefore, Hamming code encoding may encounter more serious error-masking problem under symmetric errors.

Table I. Distribution of check symbols.

Check bit	Berger	Add	Sub
000	1(-)	1(-)	-
001	4(2)	2(2)	1(-)
010	6(2)	3(2)	2(2)
011	4(2)	4(2)	3(2)
100	1(-)	3(2)	4(2)
101	-	2(2)	3(2)
110	-	1(-)	2(2)
111	-	-	1(-)

In terms of the hardware overhead due to checkers, the proposed method is close to Hamming code encoding. In both schemes, the checkers consist of full adders, and the numbers of required checkers for several different vector lengths are listed in Table II. Three encoding schemes are given, Berger code, Add(3) (for k=3) and Add(4) (for k=4). Note that, in our scheme, the adders can be used to conduct either addition or subtraction, so the hardware overhead is identical for both methods. In the column under Berger, the number given in each parenthesis is the length of the corresponding check symbol.

Table II. Hardware overhead.

# of bit	Berger	Add(3)	Add(4)
6	4(3)	5	3
12	11(4)	15	13
28	26(5)	32	30
54	57(6)	59	57

On the other hand, the hardware overhead due to CSG is usually proportional to the length of check symbol. Since in our method the check symbols are usually smaller than or equal to the check symbols for Berger code, the overall hardware overhead due to our method is smaller. Note that the checker design for weight-based code is much more complicated than that of Berger code encoding, the hardware overhead due to weight-based code is always larger than the proposed method.

4. Experimental Results

In order to verify the effectiveness of the proposed method, we conduct experiments on 10 ISCAS85 benchmark circuits. The statistics of these benchmark circuits are listed in Table III.

Table III. Benchmark statistics.

Circuits	# Primary Inputs	# Primary outputs	# Total Fault Detected
C432	36	7	282582
C499	41	32	1260334
C880	60	26	167976
C1355	41	32	1547371
C1908	33	25	1402446
C2670	233	140	2934479
C3540	50	22	2121782
C5315	178	123	3773836
C6288	32	32	15745993
C7552	207	107	5934156

The experiments are executed as follows. We generate 10000 random input vectors for each circuit, and conduct fault simulation for all single stuck-at faults in the circuit under test. Whenever there are errors in the output vector, we verify if the regenerated check symbol *CS'* is the same as the fault-free check symbol. If the results are the same, the error is undetectable. Let *Ne* be the number of erroneous output vectors due to single stuck-at faults, and *Nd* be the number of erroneous output vectors that are not masked. The fault coverage then can be defined as *Nd*/*Ne*. The fault coverage for basic encoding schemes are given in Table IV. The conventions used in this Table are similar to those used in previous Tables. For encoding schemes are used: Berger code, output compression with addition (*k*=3), subtraction (*k*=3), and both (*k*=6). Note that in the last column, the method "Mix" gives a 6-bit check symbol, in which 3 bits are the same as Add(3) while the other 3 are the same as Sub(3).

Table IV. Fault coverage: basic encoding schemes.

circuits	Berger	Add(3)	Sub(3)	Mix(6)
C432	88.79(3)	80.80	90.77	98.74
C499	90.37(6)	86.53	91.84	97.07
C880	85.67(5)	79.98	87.51	96.84
C1355	90.07(6)	84.92	89.20	96.20
C1908	87.31(5)	76.65	84.46	94.69
C2670	86.20(8)	80.25	85.23	95.54
C3540	86.54(5)	70.47	82.26	93.89
C5315	84.85(7)	78.94	86.10	95.81
C6288	86.35(6)	73.61	82.69	95.48
C7552	96.25(7)	71.13	78.05	93.38
average	88.24	78.33	85.81	95.77

It can be seen from table IV that Subtraction gives better than Addition operation. However, the fault coverage achieved in either Add(3) or Sub(3) is worse than that of Berger code's, and the fault coverage tends to become lower as the vector length getting larger. The reason is simple: too many level of data compression creates more alias problems. On the other hand, the fault coverage achieved in the last column is much higher than either Add(3) or Sub(3), which implies only a few errors are masked in both cases.

Higher fault coverage can be achieved with longer check symbols. If we increase the length of check symbol from 3 to 4 in our methods, the number of adder stages is reduced by 1 and thus the fault coverage becomes higher. The results are shown in Table V. It can be seen that the fault coverage becomes higher in all cases for k=4.

Table V. Fault coverage: longer check symbols.

circuits	Add(3)	Sub(3)	Add(4)	Sub(4)	Mix(6)	Mix(8)
C432	80.80	90.77	88.82	93.61	98.74	99.46
C499	86.53	91.84	96.65	97.00	97.07	98.99
C880	79.98	87.51	84.83	90.45	96.84	97.92
C1355	84.92	89.20	96.84	96.91	96.20	99.09
C1908	76.65	84.46	84.64	88.18	94.69	96.88
C2670	80.25	85.23	87.45	91.25	95.54	97.23
C3540	70.47	82.26	78.24	85.55	93.89	96.05
C5315	78.94	86.10	91.14	93.83	95.81	98.61
C6288	73.61	82.69	88.50	90.52	95.48	98.83
C7552	71.13	78.05	84.69	86.38	93.38	96.67
average	78.33	85.81	88.18	91.37	95.77	97.98

Finally, in Table VI we compare our design methods with previous methods, including encoding with Berger code and Weight-based code [11]. Note that for the weight-based code we select weights according to those given in [11]; however, the weights are randomly assigned to outputs. It can be seen that Sub(4) achieves a higher fault coverage than Berger code encoding, while the number of bits in check symbols is smaller in most cases. However, the fault coverage may not be good enough. Mix(6) provides a fault coverage close to that of weight-based code with fewer check symbols and simpler checker design.

Table VI: Comparison.

circuits	Mix(6)	Sub(4)	Berger	weight[11]
C432	98.74	93.61	88.79(3)	97.70(5)
C499	97.07	97.00	90.37(6)	97.11(7)
C880	96.84	90.45	85.67(5)	94.46(6)
C1355	96.20	96.91	90.07(6)	96.89(6)
C1908	94.69	88.18	87.31(5)	96.97(7)
C2670	95.54	91.25	86.20(8)	93.54(8)
C3540	93.89	85.55	86.54(5)	96.77(7)
C5315	95.81	93.83	84.85(7)	94.53(9)
C6288	95.48	90.52	86.35(6)	91.16(6)
C7552	93.38	86.38	96.25(7)	99.05(9)
average	95.77	91.37	88.24	95.82

5. Concluding Remarks

In this paper, we presented a new design method for self-checking circuits. This concurrent error-checking scheme employs a simple and structured checker, which is usually smaller and easy to implement. This method also provides a trade-off between fault coverage and area overhead. Compared with previous methods, this method can achieve a higher fault coverage with lower area overhead.

References

[1] X. Castillo, S.R. McConnel, and D.P. Siewiorek, “Derivation and calibration of a transient error reliability model,” *IEEE Trans. Comput.*, vol. C-31, pp. 658-671, July 1982.

[2] Y. Savaria, N.C. Rumin, J.F. Hayes, and V.K. Agarwal, “Soft-error filtering: A solution to the reliability problem of future VLSI digital circuits,” *IEEE Proc.*, vol. 74, no. 5, pp. 669-683, May 1986.

[3] M.M. Yen, W.K. Fuchs, and J.A. Abraham, “Designing for concurrent error detection in VLSI: Application to a microprogram control-unit,” *IEEE J. Solid-State Circuits*, vol. SC-22, pp. 595-605, Aug. 1987.

[4] G.P. Mak, J.A. Abraham, and E.S. Davidson, “The design of PLAs with concurrent error detection,” in *Proc. Int’l Symp. Fault-Tolerant Comput.*, June 1982, pp. 303-310.

[5] N. K. Jha, and S.Wang, “Design and synthesis of self-checking VLSI circuits,” *IEEE Trans. Computer-Aided Design,* Vol. 12, No.6, pp. 878-887, Jun. 1993.

[6] K. De, C. Natarajan, D. Nair, and P. Banerjee, “RSYN: A system for automated synthesis of reliable multilevel circuits,” *IEEE Trans. VLSI Systems,* pp. 186-195, Jun. 1994.

[7] P.P. Saposhnikov, A. Morosov, and M. Goessel, “A new design method for self-checking unidirectional combinational circuits,” *J. Electronic Testing: Theory and Application*, pp. 41-53, 1998.

[8] N. A. Touba, and E.J. McCluskey, “Logic synthesis techniques for reduced area implementation of multilevel circuits with concurrent error detection,” *Int’l Conf. Computer-Aided Design*, pp. 651-654, 1994.

[9] N.A. Touba, and E.J. McCluskey, “Logic synthesis of multilevel circuits with concurrent error detection, ” *IEEE Transactions on Computer-Aided Design*, vol. 16, no. 7, pp. 783-789, Jul. 1997.

[10] D. Das and N. A. Touba, “Synthesis of low-cost concurrent error detection based on bose-lin codes,” *Proc. VLSI Test Symp.*, pp. 309-315, 1998.

[11] D. Das, and N. A. Touba, “Weight-based codes and their application to concurrent error detection of multilevel circuits”, *Proc. of VLSI Test Symp.*, pp. 370-376, 1999.

[12] W.C. Carter and P.R. Schneider, “Design of dynamically checked computers,” in Proc. IFIP’68, vol. 2, Aug. 1968, pp. 878-883.

[13] D.A. Anderson and G. Metze, “Design of totally self-checking circuits for *m*-out-of-*n* codes,” *IEEE Trans. Comput.* vol. C-22, pp. 263-269, Mar. 1973.

[14] J. M. Berger, “A note on error detection codes for asymmetric binary channels”, *Inform. Contr.*, pp 68-73, Mar. 1961.