Design and Synthesis of Self-Checking VLSI Circuits

Niraj K. Jha, Member, IEEE, and Sying-Jyan Wang

Abstract—Self-checking circuits 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 output vectors. The checker has the ability to expose its own faults as well. The functional circuit can be either combinational or sequential. 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.

Although much effort has been concentrated on the design of self-checking checkers by previous researchers, very few results have been presented for the design of self-checking functional circuits. In this paper, we explore methods for the cost-effective design of combinational and sequential self-checking functional circuits and checkers. The area overhead for all proposed design alternatives is studied in detail.

I. INTRODUCTION

TECHNOLOGICAL advances have increased drastically the complexity of integrated circuits that can be realized on a single chip, and the trend is likely to continue. Although the advantages of such a rapid advancement are obvious, we do have to deal effectively with some of the side effects, such as the increase in the number of transient faults. It is well known that transient faults are the predominant cause of system failures [1]–[3]. With the possibility of reduced voltage levels for VLSI circuits and the resultant decrease in noise margins, system susceptibility to transient faults is likely to increase [3]. Thus we have to design circuits and systems that can expose such faults.

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 vectors to determine 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 most important concept developed for self-checking circuits is the totally self-checking (TSC) concept [4], [5]. The TSC concept was generalized to the path fault-secure (PFS) and strongly fault-secure (SFS) concepts [6]. These concepts are defined in Section II.

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 integrated circuits [3], [7]–[10]. Such an error is said to occur when there are multiple transitions in either the 0 → 1 direction or the 1 → 0 direction, but not both. For example, in [9] it was shown that a stuck-at fault, cross-point fault, or a short in an MOS programmable logic array (PLA) or read-only memory causes a unidirectional error at its outputs. Even for random logic, it is easy to ensure by simple design techniques that most single faults can cause only a unidirectional error. Many codes have been proposed for detecting such errors [10]–[16], [22], [26], [27]. A nice mathematical framework for dealing with such errors is given in [10].

A lot of work has been done in the area of self-checking checker design for different codes. However, not as much attention has been paid to the design of functional circuits. Self-checking combinational functional circuit design methods were presented in [17] and [6]. Results in the area of self-checking sequential functional circuit design were presented in [18]–[20]. Applications of the self-checking concept to microprogram control units and PLA’s were presented in [7], [3], and [9]. A theory of TSC system design was presented in [21]. In most previous works on functional circuits, usually some conditions are given which need to be satisfied to obtain a self-checking design. However, no attempt is made to evaluate the practicality of the design in terms of area overhead, etc. For example, the area overhead needed for a self-checking sequential functional circuit design derived from some existing methods can be quite substantial.

In this paper, we are concerned with deriving methods for cost-effective design of combinational and sequential functional circuits and checkers. In Section II, we describe briefly some previous results that will be used herein. The basic self-checking circuit and system structures are given in Section III. Next, in Section IV, we evaluate various synthesis methodologies by applying them to some benchmarks. Concluding remarks are given in Section V.

II. PRELIMINARIES

In this section, first we will review some unidirectional error-detecting codes that will be used in the following sections. Then we will discuss properties that define a self-checking circuit.
2.1. Systematic and Separable Codes

First, we will make a distinction between systematic and separable codes. Suppose that the information symbol has \( k \) bits and the appended check symbol has \( r \) bits. Thus, a codeword has \( n = k + r \) bits.

**Definition 1:** A set of binary \( n \)-tuples \( C \) is called a systematic code if 1) it contains \( 2^k \) \( n \)-tuples, \( 1 \leq k \leq n \), and 2) \( C = \{ x | x = IS, CS \} \), where \( |IS| = k \) and \( |CS| = r \), and each one of the \( 2^k \) information symbols appears as part of some codeword.

**Definition 2:** A set of binary \( n \)-tuples \( C \) is called a separable code if 1) it contains \( s < 2^k \) \( n \)-tuples, \( 1 \leq k \leq n \), and 2) \( C = \{ x | x = IS, CS \} \), where \( |IS| = k \) and \( |CS| = r \), and each one of the \( s \) information symbols appears as part of some codeword.

The Berger code [22] is an optimal systematic all-unidirectional error-detecting (AUED) code. For such a code, \( r = \lceil \log_2 (k+1) \rceil \). This code is formed as follows: a binary number corresponding to the number of 1’s in the information symbol is obtained, and the binary complement of each bit in this number is taken; the resulting binary number forms the check symbol. Sometimes AUED codes are also called unordered. TSC checkers for the Berger code were presented in [23]-[25] and [40].

In a systematic code in which the number of information bits is \( k \), it is assumed that all \( 2^k \) possible information symbols are present. However, more often than not, a functional circuit with \( k \) outputs does not produce all the \( 2^k \) output patterns. In such cases, it is possible to use a separable AUED code [26], [16] for encoding purposes. A separable code has redundancy no worse than a systematic code with the same capability; frequently its redundancy is less.

2.2. The m-out-of-n Code

The \( m \)-out-of-\( n \) code [27] is an optimal AUED code. In such a code, all codewords have exactly \( m \) 1's and \( (n-m) \) 0's. For this reason, an \( m \)-out-of-\( n \) code is also called an \( m \)-hot code. Such codes are "nonsystematic" because the information bits cannot be separated from the check bits. A lot of work has already been done in the area of TSC checker design for \( m \)-out-of-\( n \) codes [5], [28]-[31].

2.3. Self-Checking Circuits

The following definitions can be used to describe a TSC system [4], [5]. \( F \) denotes the set of faults, and \( G \) is the network in these definitions.

**Definition 3:** \( G \) is self-testing with respect to \( F \) if for every fault in \( F \) the circuit produces an output noncodeword for at least one input codeword.

**Definition 4:** \( G \) is fault-secure with respect to \( F \) if for every fault in \( F \) the circuit never produces an incorrect output codeword for any input codeword.

**Definition 5:** \( G \) is TSC if it is both self-testing and fault secure.

**Definition 6:** \( G \) is code-disjoint if it maps input codewords (noncodewords) to output codewords (noncode-words).

**Definition 7:** \( G \) is a TSC checker if it is self-testing, fault-secure, and code-disjoint.

It has been pointed out in [32] that the fault-secure property is not really necessary for TSC checkers. Conversely, most checkers, which are designed to be only self-testing and code-disjoint, can also be found to be automatically fault-secure. Thus, in many cases we get the fault-secure property in the checker for free.

The concept of TSC functional circuits was generalized to SFS functional circuits in [6].

**Definition 8:** \( G \) is SFS with respect to \( F \) if for every fault in \( F \), either 1) the circuit is self-testing and fault-secure, or 2) the circuit is fault-secure, and if another fault from \( F \) occurs in \( G \) then either property 1 or 2 is true for the fault sequence.

The path fault-secure property also was presented in [6]. This property is defined in terms of network structures and codes. If a circuit is PFS, a modeled fault can result only in the correct codeword or a noncodeword at the outputs, never an incorrect codeword. The following theorems were proved for the PFS property [6].

**Theorem 1:** If \( G \) is PFS with respect to \( F \), it is SFS with respect to \( F \).

**Theorem 2:** If \( G \) is PFS with respect to \( F \) and input code space \( A \), then \( G \) is also PFS with respect to \( F \) for the input code space \( A \subseteq A' \).

**Theorem 3:** Any inverter-free network with an unordered output code space is PFS with respect to unidirectional stuck-at faults.

III. Self-Checking Circuit and System Structures

In this section we use the concepts introduced in the previous section to derive self-checking circuit and system structures.

3.1. Self-Checking Combinational Functional Circuits

A circuit, which has been designed to be PFS for some input code space, may actually receive a subset of the input code space under normal operation when it is placed in an actual system environment. According to Theorems 1 and 2, the circuit remains PFS and, hence, SFS. However, if a fault is not detectable by the received input code space, then the circuit can be simplified by removing the line or gate corresponding to the fault. For example, if a stuck-at 0 (stuck-at 1) fault at the input of an AND gate is undetectable by any normally received input codeword then the gate (line) can be removed. However, this process can be time-consuming.

Theorem 3 is strictly applicable to totally inverter-free circuits. It is well known that such a circuit can be obtained if AUED encoding is used for both the inputs and outputs [33], [6]. However, in general, such an approach may be overly restrictive and may result in much more area overhead than necessary. We extend this approach as
follows. We encode the output functions of the functional circuit in a unidirectional error-detecting code, but the inputs are not encoded. We then use logic optimization tools to obtain a realization of these functions which has inverters only at the primary inputs. In other words, the internal part of the circuit consists of only AND and OR gates (the circuit can have any number of levels), as shown in Fig. 1. In fact, even if the internal part of the circuit contains inverters, NAND gates or NOR gates, but the circuit can be reduced to a circuit shown in Fig. 1 only through the use of De Morgan’s theorem, then the realization is still acceptable. However, such a circuit can be guaranteed only to the PFS with respect to internal single stuck-at faults, not unidirectional stuck-at faults. Note that, in general, the output functions of a functional circuit need not be binate in each of the input variables. In Fig. 1, the outputs have been assumed to be binate in \( x_1, x_2, \ldots, x_u \), but unate in \( x_{u+1}, \ldots, x_p \).

Consider any single stuck-at fault in the circuit in Fig. 1 except those at the stems of the primary inputs \( x_1, \ldots, x_u \). Define the inversion parity of a path to be the number of inversions on the path modulo 2. For the above-mentioned faults, every path from the fault site to any circuit output has the same inversion parity. Thus these faults can result only in unidirectional errors at the outputs. Therefore, the output vector would be a noncodeword, which will be detected by the checker monitoring the outputs. This means that the circuit will be PFS and, hence, SFS with respect to the aforementioned faults. When this functional circuit is embedded in a system, it may receive its inputs from another functional circuit. The checker that monitors the outputs of the preceding functional circuit can also be made to detect the stuck-at faults on the stems of primary inputs \( x_1, \ldots, x_u \) of the functional circuit under question; this will become clearer in Subsection 3.3.

Now suppose that the functional circuit that is to be synthesized is not fed by any other functional circuit even when placed in a system. For such cases, one solution, as mentioned earlier, is to use AUED encoding for both inputs and outputs. Such a circuit is PFS with respect to all single and unidirectional stuck-at faults. Note that there is no need for an input checker. However, the disadvantage of a realization such as this is the need for extra primary inputs and most likely a significant increase in the area required for its implementation. A designer may wish to consider if even for this case the extra overhead over a circuit realization given in Fig. 1 is beneficial, considering that the extra protection is not that much, especially for VLSI functional circuits.

### 3.2. Self-Checking Sequential Functional Circuits

Previous works in the area of self-checking sequential functional circuits have been presented in [18]-[20]. However, these methods can be impractical when applied to VLSI circuits. The primary reasons are unacceptably high area overhead and/or the difficulty of satisfying the conditions required for making the circuits self-checking.

An alternative approach for Mealy machines is presented here. It can be trivially extended to Moore machines. Consider the Mealy machine shown in Fig. 2. In this machine the next state functions and the output functions are encoded separately. Suppose the number of states in the state table of the sequential machine that has to be synthesized is \( s \). If the next state functions are encoded in the \( m \)-out-of-\( n \) code, then the number of state variables or flip-flops, \( n \), must satisfy the following condition: \( \binom{n}{m} \geq s \). Clearly, to reduce the number of flip-flops, we have to choose the smallest \( n \) such that the preceding condition is satisfied, and this leads to the choice of \( m = \lceil n/2 \rceil \). However, reducing the number \( m \) usually decreases the size of the combinational logic of the sequential circuit, as we will see in Section IV. Therefore, the choice of \( m \) is a tradeoff between reducing the number of flip-flops and reducing the size of the combinational logic.

The output functions are encoded in an AUED code. Using arguments from [33] it can be shown that, with the above encoding, the circuit outputs and the next state outputs will be unate in the state variables, as shown in Fig. 2. The inputs need not be encoded if they are obtained from a preceding functional circuit (combinational or sequential). The arguments for doing this are similar to those given in Subsection 3.1. However, if the functional circuit is not fed by any other functional circuit, then the designer may or may not wish to use input encoding, as explained earlier.
There are two TSC checkers in this circuit to monitor the circuit outputs; one checks the output functions, the other checks the next state functions. The outputs of these two checkers are then fed to a two-rail checker to produce the final checker outputs. Any detectable stuck-at fault, excluding those at the primary input stems of $x_1, \cdots, x_n$, in Fig. 2, results in a unidirectional error at the circuit outputs and/or next state lines, which is caught by at least one of the checkers. The stuck-at faults at the primary input stems are detected by the checker, which checks the preceding functional circuits in the system. If there is no functional circuit preceding it, and AUED encoding is used for the primary inputs, then it can be shown that the output functions and next state functions will be unate in the primary inputs. In such a case, any stuck-at fault can cause only unidirectional errors. The preceding arguments can also be extended to PLA implementations. Thus we see that the functional circuit is PFS and, hence, SFS. This can be reduced to a TSC circuit if all gates and lines corresponding to stuck-at faults not detected by input codewords are removed. However, as mentioned earlier, this can be time-consuming.

3.3. Self-Checking Systems

The area of design and synthesis of self-checking systems has received scant attention from researchers in the past. Applications of the self-checking property to microprogram control units were discussed in [7] and [3]. A theory of TSC systems was given in [21]. However, the results presented pertain more to analysis rather than synthesis of self-checking systems. Furthermore, it may not always be possible to meet or even check for the conditions outlined in [21] especially for moderately large systems.

The design and synthesis methods for self-checking combinational and sequential circuits were presented in the previous two subsections. These are the building blocks of a system. Next we need to design the system from these building blocks. Consider the system in Fig. 3. In this system, functional circuits 1 and 2 are combinational, whereas functional circuit 3 is sequential. We have assumed that systematic or separable codes have been used to encode the output functions of the functional circuits; however, nonsystematic codes could also be used. $TRC_2$ refers to a two-variable two-rail checker. The outputs of checkers 1, 2, 3, and 4 are reduced ultimately to only two outputs, $z_1$ and $z_2$, with the help of $TRC_2$'s. From functional circuits 1 and 2, only the information bits go to functional circuit 3. In general, functional circuit 3 will require the complements of its primary inputs too (inverters are not shown in Fig. 3, but are implicit). Therefore, the branch of information bit lines that goes to checker 1 or 2 should be derived from a point beyond where these lines are fed to the inverters. Thus, we ensure that stuck-at faults on the stem and all branches of these lines can be detected.

IV. RESULTS

In this section, we give results for various synthesis methodologies. The methodologies are applied to several sequential MCNC benchmarks whose statistics are given in Table I. For each machine in this table, the number of inputs ($#inp$), the number of outputs ($#out$), and the number of states ($#sta$) are indicated. The outputs and the next states are encoded with AUED codes, as mentioned earlier, and then synthesized by MIS [34], a multilevel logic optimization program from Berkeley, to get the final realization.

4.1. Internally Inverter-Free Sequential Circuits Encoded with Berger Code

The first synthesis scheme follows the procedure given in Subsection 3.2. The output functions are encoded with the Berger code, while the states of the finite state machines are encoded with the $m$-out-of-$n$ code with all $m$
and \( n \) that satisfy the following conditions: \( \binom{n}{s} \geq s \) and \( m \leq \lceil n/2 \rceil \), where \( s \) is the number of states. The functional circuits are optimized by a script that strictly uses algebraic factorization without complement; i.e., the resulting circuits have inverters only at the primary inputs, but not inside them, as shown in Fig. 2. However, when such a script is used to optimize a circuit, the results are usually slightly inferior to a script that includes Boolean optimization.

Consider, for example, the state table shown in Fig. 4(a). With the given state assignment, the combinational logic of the sequential circuit without any self-checking properties is as shown. However, when 1-hot encoding for the states and Berger encoding for the outputs are used, then the self-checking circuit is as shown in Fig. 4(b).

Let us next consider the FSM benchmarks, which are first converted to KISS2 format [35]. With an \( m \)-out-of-\( n \) (or, equivalently, \( m \)-hot) state assignment, the circuit outputs and next state outputs can be made unate in state variables by specifying the 0's in the present state assignment to be "don't care." For instance, if the present state is 11 000, then it should be specified as 11- ..., where the dash denotes a don't care. If this is not done, MIS does not necessarily find the solution that makes the outputs unate in the present state inputs. The functional circuit outputs are then encoded with the Berger code. The literal-counts of the combinational logic of the self-checking sequential circuits synthesized by our method are given in Table II. For each benchmark, we tried all possible \( m \)-hot codes, and for each \( m \)-hot code, two circuits were synthesized. In the first circuit, the output functions are encoded with Berger code, whereas in the second one they are not. Both of them are optimized with the algebraic script so that any internal stuck-at fault will cause only unidirectional errors at the outputs. In the last column, the target machine's states are assigned by MUSTANG [36], which gives a state assignment with \( \lceil \log_2 s \rceil \) bits; the circuit is then fully optimized (i.e., Boolean optimization is performed). There are two options in MUSTANG, one being fan-in-oriented and the other fan-out-oriented. Synthesis was done based on both options, and the better of the two results is given in Table II. In this table the area occupied by the latches was not taken into account because first we want to draw some conclusions from these results about the overheads involved only in the combinational logic. However, when we report the total literal-count of the functional circuit and the checker, we will also take into account the area of the latches.

It is interesting to note that, as the state encoding moves from 1-hot to 2-hot, etc., the literal-count of the combinational logic of the self-checking sequential circuit increases as well. In many cases, the 1-hot code is better than MUSTANG in terms of literal-count. However, the 1-hot code may not always be practical for the larger machines (with more states) since the number of required latches increases so fast that the gain in the literal-count by the 1-hot code may be more than compensated by the increase in the area due to the extra latches, as we will
The 2- and 3-hot codes are usually reasonably close to the Berger encoding is $\log_2 (k + 1)$, which makes them a good choice. For larger machines, the 2-hot encoding can give better results than the original circuit has. Hence, for these circuits the percent overhead can be large. However, if there are inverters inside the circuit, then there might be bidirectional errors at the outputs. But, it is still possible that a bidirectional error can be detected by either the Berger-encoded circuit outputs or the m-out-of-n encoded next state assignments is close to 1, as one may have naively expected. The reason for reporting normalized overhead, as opposed to simply the percent overhead, is that it allows us to see the trend. All the benchmark circuits are reasonably small. Hence, for these circuits the percent overhead can be large. However, if the preceding trend holds for larger sequential circuits, then we can see that the percent overhead for such circuits can go down drastically, since the ratio $\log_2 (k + 1) / k + p$ decreases as the value of $k$ increases.

### 4.2. Fully Optimized Sequential Circuits Encoded with Berger Code

If a circuit is optimized with algebraic factorization, then any internal stuck-at fault will cause only unidirectional errors at the outputs. However, if there are inverters inside the circuit, then there might be bidirectional errors at the outputs. But, it is still possible that a bidirectional error can be detected by either the Berger-encoded circuit outputs or the m-out-of-n encoded next state outputs. Therefore, if the fully optimized circuit has a smaller overhead, yet if a high enough percentage of faults are detected by the above codes, then it may be worthwhile using full optimization. In this subsection, we report results on the synthesis of the self-checking circuits obtained by full optimization, and on the probability that a fault will invalidate the above codes (i.e., change one codeword into another). In Table IV we give the literal-counts for the benchmark sequential circuits.

**TABLE II**

<table>
<thead>
<tr>
<th>CKT</th>
<th>1-hot</th>
<th>2-hot</th>
<th>3-hot</th>
<th>4-hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbsse</td>
<td>169</td>
<td>126</td>
<td>217</td>
<td>186</td>
</tr>
<tr>
<td>cse</td>
<td>266</td>
<td>208</td>
<td>319</td>
<td>278</td>
</tr>
<tr>
<td>dk14</td>
<td>175</td>
<td>146</td>
<td>235</td>
<td>196</td>
</tr>
<tr>
<td>dk15</td>
<td>141</td>
<td>116</td>
<td>169</td>
<td>137</td>
</tr>
<tr>
<td>dk16</td>
<td>238</td>
<td>201</td>
<td>355</td>
<td>321</td>
</tr>
<tr>
<td>exl</td>
<td>422</td>
<td>210</td>
<td>502</td>
<td>295</td>
</tr>
<tr>
<td>ex6</td>
<td>111</td>
<td>76</td>
<td>160</td>
<td>117</td>
</tr>
<tr>
<td>kirkman</td>
<td>137</td>
<td>101</td>
<td>173</td>
<td>124</td>
</tr>
<tr>
<td>mark1</td>
<td>106</td>
<td>92</td>
<td>152</td>
<td>136</td>
</tr>
<tr>
<td>mc</td>
<td>44</td>
<td>44</td>
<td>67</td>
<td>57</td>
</tr>
<tr>
<td>opus</td>
<td>95</td>
<td>74</td>
<td>134</td>
<td>108</td>
</tr>
<tr>
<td>planet</td>
<td>493</td>
<td>356</td>
<td>667</td>
<td>533</td>
</tr>
<tr>
<td>sand</td>
<td>518</td>
<td>449</td>
<td>641</td>
<td>562</td>
</tr>
<tr>
<td>scf</td>
<td>698</td>
<td>556</td>
<td>979</td>
<td>830</td>
</tr>
<tr>
<td>sse</td>
<td>169</td>
<td>126</td>
<td>217</td>
<td>186</td>
</tr>
<tr>
<td>styr</td>
<td>538</td>
<td>385</td>
<td>660</td>
<td>536</td>
</tr>
</tbody>
</table>

---: There is no need for such a code.
*: Results not available.

**TABLE III**

<table>
<thead>
<tr>
<th>CKT</th>
<th>1-hot</th>
<th>2-hot</th>
<th>3-hot</th>
<th>4-hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbsse</td>
<td>2.62</td>
<td>0.78</td>
<td>0.13</td>
<td>-</td>
</tr>
<tr>
<td>cse</td>
<td>2.14</td>
<td>0.69</td>
<td>0.70</td>
<td>-</td>
</tr>
<tr>
<td>dk14</td>
<td>0.80</td>
<td>0.67</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>dk15</td>
<td>0.65</td>
<td>0.70</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>dk16</td>
<td>2.76</td>
<td>0.58</td>
<td>0.38</td>
<td>-</td>
</tr>
<tr>
<td>ex1</td>
<td>7.87</td>
<td>2.63</td>
<td>1.94</td>
<td>-</td>
</tr>
<tr>
<td>ex4</td>
<td>2.65</td>
<td>1.38</td>
<td>1.05</td>
<td>-</td>
</tr>
<tr>
<td>ex6</td>
<td>1.43</td>
<td>1.28</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>kirkman</td>
<td>5.00</td>
<td>2.15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>mark1</td>
<td>0.91</td>
<td>0.52</td>
<td>0.66</td>
<td>-</td>
</tr>
<tr>
<td>mc</td>
<td>0.00</td>
<td>0.53</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>opus</td>
<td>0.79</td>
<td>0.88</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>planet</td>
<td>5.16</td>
<td>1.51</td>
<td>1.18</td>
<td>1.15</td>
</tr>
<tr>
<td>sse</td>
<td>1.26</td>
<td>0.49</td>
<td>0.39</td>
<td>-</td>
</tr>
<tr>
<td>sand</td>
<td>1.58</td>
<td>0.63</td>
<td>0.69</td>
<td>-</td>
</tr>
<tr>
<td>scf</td>
<td>7.53</td>
<td>2.18</td>
<td>2.48</td>
<td>2.23</td>
</tr>
<tr>
<td>styr</td>
<td>2.62</td>
<td>0.78</td>
<td>0.13</td>
<td>-</td>
</tr>
<tr>
<td>Average</td>
<td>2.76</td>
<td>1.08</td>
<td>0.93</td>
<td>1.69</td>
</tr>
</tbody>
</table>
counts for fully optimized circuits in the same way as in Table II.

Comparing Tables II and IV, it is interesting to note that fully optimized circuits sometimes have worse literal-count than the circuits optimized by the algebraic script. Actually, it seems that the algebraic script nearly always produces better results for larger circuits. The reason, however, may be that the output and the next state functions are unate in the present state variables, and this property may be better exploited by algebraic operations.

Table V is similar to Table III, and it shows the normalized overhead for the fully optimized circuits.

The next thing we want to know is the probability that a fault will be detected if the method described in this section is used. To obtain this probability, we generate a test pattern for each stuck-at fault in the combinational logic of the synthesized circuit, and see if the errors at the output are detectable by the codes. We used the test pattern generation program STALLION [37] with some modifications. The result shown in Table VI is the probability that a fault is not detected. The average probability ranges from 4 to 7% for 2-, 3-, and 4-hot codes. Depending on the application, such a probability may or may not be considered too high.

### 4.3. TSC Checkers

In this subsection, we describe briefly the checkers for the codes used herein.

TSC checkers for the Berger code were presented in [23]–[25] and [40]. We consider the area-efficient method given in [24] next. The checkers presented in [24] consist of half-adders, full-adders, and two-rail checkers. Suppose the number of information bits in a codeword is \( k \), and the number of checkbits is \( r \). Then the number of full-adders is \( k \cdot \lceil \log_2 k \rceil - 1 \), the number of half-adders is upper bounded by \( \lceil \log_2 k \rceil \cdot \lceil \log_2 k \rceil \), and the number of two-variable two-rail checkers (TRC\(_2\)'s) required to form an \( r \)-variable two-rail checker is \( \lceil \log_2 (k + 1) \rceil - 1 \). The least literal-counts for a full-adder, a half-adder, and a TRC\(_2\) checker among various designs we considered are 13, 6, and 8, respectively. Therefore, if \( l.c. \) is the literal-count of a Berger code checker, the bound on it is given by

\[
13(k - \lceil \log_2 k \rceil - 1) + 8(\lceil \log_2 (k + 1) \rceil - 1) \\
\leq l.c. \\n13(k - \lceil \log_2 k \rceil - 1) + 6(\lceil \log_2 k \rceil) \\
+ 8(\lceil \log_2 (k + 1) \rceil - 1).
\]

There are many TSC \( m \)-out-of-\( n \) code checker designs in the literature [5], [28]–[31], from which the designer...
can choose the best. Usually these checkers are rather small because of the small values of $n$, except when 1-hot code is used. In the following discussion, for each $m$-hot code, we choose the design with the least literal-count.

The final circuit size depends on the size of the combinational logic of the sequential circuit, latches, and checkers. The actual area occupied by the latches may be a decisive factor in choosing among the different $rn$-hot checkers. The actual area occupied by the latches may be binational logic of the sequential circuit, latches, and the two checkers) for all benchmark ex-

The functions of the CSCG are derived as follows. For example, let $r=3$, $k=2$, $G=3$, therefore, $r$ is chosen to be $\lceil \log_2 (G + 1) \rceil$. The $r$-bit check symbols are assigned to the different groups in ascending order (i.e., the group with a larger zero-count is assigned a larger check symbol). For example, let $k=9$ and the different groups present have zero-counts 2, 5, 6, 7, and 9. Then $G = 6$ and $r = 3$. Therefore, one possible assignment would be to use the check symbols 110, 101, 100, 011, 010, 001, respectively, for the information symbols with zero-counts 9, 7, 6, 5, 2, 1.

A self-checking circuit based on the preceding encoding technique is shown in Fig. 5. The functional circuit consists of two subcircuits: the information symbol generator (ISG) and the check symbol generator (CSG). These two subcircuits do not share any logic. The checker consists of a check symbol complement generator (CSCG), which generates the bit-by-bit complement of the check symbol in the fault-free case, and an $r$-variable two-rail checker ($T_{RC}$). The structure of this checker for the separable code is the same as the structure frequently used for systematic codes [23].

The functions of the CSCG are derived as follows. For all the information symbols that have the same zero-count into one group. Suppose there are $G$ such groups. Obviously, $1 \leq g \leq k + 1$. Let $r$ be the number of checkbits. We do not use the check symbol with $r = 1$’s. Therefore, $r$ is chosen to be $\lceil \log_2 (G + 1) \rceil$. The $r$-bit check symbols are assigned to the different groups in ascending order (i.e., the group with a larger zero-count is assigned a larger check symbol). For example, let $k=9$ and the different groups present have zero-counts 2, 5, 6, 7, and 9. Then $G = 6$ and $r = 3$. Therefore, one possible assignment would be to use the check symbols 110, 101, 100, 011, 010, 001, respectively, for the information symbols with zero-counts 9, 7, 6, 5, 2, 1.

A self-checking circuit based on the preceding encoding technique is shown in Fig. 5. The functional circuit consists of two subcircuits: the information symbol generator (ISG) and the check symbol generator (CSG). These two subcircuits do not share any logic. The checker consists of a check symbol complement generator (CSCG), which generates the bit-by-bit complement of the check symbol in the fault-free case, and an $r$-variable two-rail checker ($T_{RC}$). The structure of this checker for the separable code is the same as the structure frequently used for systematic codes [23].
observation, it is easy to show that the checker is code-disjoint.

The fact that sharing of logic is not allowed between ISG and CSG tends to increase the area of the functional circuit. However, since there may be fewer bits in the check symbol and the area of the checker may be less, the overall area for a self-checking circuit based on a separable code may be less than the area of a self-checking circuit based on a systematic code. The preceding approach can obviously be extended easily to sequential circuits, as before. In this extension, the ISG can be combined with the next state logic, but the CSG has to be synthesized separately.

The results for some large FSM sequential benchmark machines are given in Table VIII. As before, the states of each machine are encoded with all possible m-hot codes; the number given in the corresponding column is the literal-count for the combinational logic of the sequential circuit whose outputs are encoded with separable codes. The last column gives the literal-count of the separable code checker, which includes CSCG and TRC.

Next, in Table IX we give the total literal-count of the self-checking circuit based on separable codes. This includes the literal-counts for the functional circuit (including both the combinational logic and latches), the separable code checker and the m-out-of-n code checker. The literal-counts for the duplicate-and-compare scheme are also listed as a reference. By looking at the results in Table VII, we find that the literal-counts for the self-checking circuits based on separable codes are better for mark and scf, and worse for ex1 and planet compared with the literal-counts for self-checking circuits based on the Berger code, which is a systematic code.

V. CONCLUSION

In this paper we presented gate-level self-checking circuit, system, and checker synthesis methodologies. The overhead due to the proposed methodologies was evaluated by synthesizing self-checking circuits for MCNC benchmarks.

From the experimental results we found that, if the states of a sequential circuit are encoded with an m-hot code, the size of the combinational logic usually increases as m increases. However, when the area of the whole self-checking circuit is taken into account (including that of the latches and checkers), then the 2-hot code seems to give the best result in most cases. Second, since the normalized overhead of the combinational logic is close to 1 on the average (at least for the 2- and 3-hot codes), it means that one can expect the overhead introduced in the circuit to be close to \[ \log_2(k + 1) / k + p \], where k and p are the number of primary outputs and next state lines, respectively. Therefore, the overhead due to Berger encoding should, in general, decrease as the size of the circuit becomes larger.

Self-checking systems are of great value for ensuring reliability of computations. Therefore, we hope that our results will help designers of such systems choose from among the different approaches.

REFERENCES


Niraj K. Jha (S'85-M'86) received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India in 1981, the M.S. degree in electrical engineering from S.U.N.Y. at Stony Brook, NY, in 1982, and the Ph.D. degree in electrical engineering from university of Illinois, Urbana, IL, in 1985. He is currently an Associate Professor of Electrical Engineering at Princeton University. From 1985 to 1987 he was an Assistant Professor of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. He has served as the Program Chairman of the 1992 Workshop on Fault-Tolerant Parallel and Distributed Systems. He has also served on the program committees for the IEEE International Conference on Computer Design in 1988, 1989, and 1990, for the IEEE International Symposium on Fault-Tolerant Computing in 1990 and 1991, and the International Conference on VLSI Design in 1993. He has co-authored a book titled Testing and Reliable Design of CMOS Circuits. He has authored or co-authored over 80 technical papers. His research interests include digital system testing, fault-tolerant computing, computer-aided design of integrated circuits, and parallel processing. He is the recipient of the AT&T special-purpose grant award and the NEC Preceptorship award.

Sying-Jyan Wang received the B.S. degree in electronics engineering from the National Taiwan University, Taiwan, China, in 1984, and the Ph.D. degree in electrical engineering from Princeton University in 1992. From 1984 to 1986 he was an R.O.T.C. officer in the Chinese Air Force, Taiwan. From 1986 to 1987 he was a teaching assistant in the Department of Electrical Engineering, National Taiwan University. He is currently an Associate Professor of Institute of Information Science at National Chung Hsing University. His research interests include fault-tolerant computing, digital system testing, and communication architectures.