

Systematic AUED Codes for Self-Checking Architectures

Donatella Sciuto [†], Cristina Silvano [‡], Renato Stefanelli [†]

[†]Politecnico di Milano, Dipartimento di Elettronica e Informazione
P.zza L. da Vinci 32, I-20133 Milano (ITALY)

[‡]Università di Brescia, Dipartimento di Elettronica per l'Automazione,
Via Branze 38, I-23125 Brescia (ITALY).

Abstract

Encoding techniques and dedicated self-checking architectures can be conveniently adopted in VLSI design to increase fault detection. Area overhead and speed penalty may be traded-off with fault detection capabilities. Aim of this work is to define a class of systematic all-unidirectional error-detecting (AUED) codes suitable for self-checking architectures for multiple output combinational circuits. A class of systematic AUED codes is proposed along with a logic synthesis algorithm to derive the redundant functions directly from the primary inputs. If compared with Berger codes, the proposed encoding techniques require greater output redundancy but provide performance optimization in terms of both transition delays and area overhead.

I. Introduction

With the increasing complexity of digital circuits and systems, reliability requirements have become one of the most important issues. To achieve reliability, error detecting and correcting codes, adding redundancy to data bits, have found wide applicability to VLSI circuits.

As reported in [1, 2], unidirectional errors are the most dominant type of errors in CMOS devices. A unidirectional error is said to occur when all errors are either $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions, but not both, in any given output data word. The all-unidirectional error-detecting (AUED) codes can detect all possible unidirectional errors. Both systematic and non-systematic codes are known to detect all unidirectional errors. In general, systematic or separable codes, for which the redundant bits are appended to the information bits, are preferred, so that the encoding/decoding and data manipulation can be performed in parallel. The m -out-of- n or constant-weight codes are non-systematic AUED codes, while Berger codes are optimal systematic AUED codes in terms of redundant information.

Aim of this work is to provide a comprehensive technique to generate a large class of systematic AUED codes for multiple output functions. Within the code class, we define a method to select codes satisfying different criteria, such as minimum redundancy and minimum area overhead required by the encoding circuits. A new synthesis methodology has been defined to directly derive the redundant outputs from the primary inputs with the twofold purpose to reduce area and speed penalty for the redundant logic.

The paper is organized as follows. Section II describes the proposed AUED code class along with code selection criteria. The innovative synthesis method is reported in Section III, where the proposed method is also applied on a 2-bit binary adder. Finally concluding remarks and future directions for the work conclude the paper.

II. The New Class of Systematic AUED Codes

The proposed class of systematic AUED codes is defined by the following equations:

$$\begin{cases} f_k = g_k' + \sum_{r=0}^{n-1, r \neq k} g_{kr}' f_r' & k = 0, 1, \dots, n-1 \\ g_{kr} = g_{rk} \end{cases}$$

where the f_k functions are the functional outputs and the g_k and g_{kr} functions are the redundant functions. The proposed codes are *systematic AUED*, being each couple of codewords unordered [7], and the redundant functions are symmetric functions in the sense that $g_{kr} = g_{rk}$. By using the vector representation, we obtain: $F = G' + A F'$ where:

$$F = \begin{bmatrix} f_0 \\ f_1 \\ \dots \\ f_k \\ \dots \\ f_{n-1} \end{bmatrix} \quad G = \begin{bmatrix} g_0 \\ g_1 \\ \dots \\ g_k \\ \dots \\ g_{n-1} \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & g_{01}' & g_{02}' & \dots & g_{0k}' & \dots & g_{0n-1}' \\ g_{01}' & 0 & g_{12}' & \dots & g_{1k}' & \dots & g_{1n-1}' \\ g_{02}' & g_{12}' & 0 & \dots & g_{2k}' & \dots & g_{2n-1}' \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ g_{0k}' & g_{1k}' & g_{2k}' & \dots & 0 & \dots & g_{kn-1}' \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ g_{0n-1}' & g_{1n-1}' & g_{2n-1}' & \dots & g_{kn-1}' & \dots & 0 \end{bmatrix}$$

The equation representing the code is: $(f_0 \oplus S_0) (f_1 \oplus S_1) \dots (f_k \oplus S_k) \dots (f_{n-1} \oplus S_{n-1}) = 1$

where:

$$S_k = g_k' + \sum_{r=0}^{n-1, r \neq k} g_{kr}' f_r' = (g_k \prod_{r=0}^{n-1, r \neq k} (g_{kr} + f_r))' \quad k = 0, 1, \dots, n-1$$

From the code equations, we can extract a large set of conditions that can be expressed in a table of codewords for n -output functions (see for example Tables I, II and III for $n = 1, 2, 3$ respectively). Basically, each row in the tables represents a codeword and the g_k and g_{kr} terms of each row are related by a set of conditions identified with the c_k symbols. For each row, let us consider all c_i symbols with the same subscript i . For these symbols, the following condition must hold: at least one of c_i must be equal to 0, while the remaining ones can be don't care symbols. For example, the symbols c_1 and c_2 in Table II mean $g_1' g_{01}' = 0$ and $g_0' g_{01}' = 0$, respectively.

We can prove that each couple of codewords is unordered for each value of the conditions c_i , therefore the code is an AUED code [1]. The demonstration of this properties is straightforward for 2- and 3- output functions, while the demonstration for the general case is reported in [7]. The total number of codes that can be derived is quite large, hence we need some criteria to identify the codes satisfying the following combined conditions: minimum number of additional functions and minimum area for the redundant logic.

f_0	g_0
0	1
1	0

Table I: Codewords for single-output functions

f_0	f_1	g_0	g_1	g_{01}
0	0	1	1	1
0	1	1	c_1	c_1
1	0	c_2	1	c_2
1	1	0	0	x

Table II: Codewords for 2-output functions

f_0	f_1	f_2	g_0	g_1	g_2	g_{01}	g_{02}	g_{12}
0	0	0	1	1	1	1	1	1
0	0	1	1	1	1	c_1	1	c_1
0	1	0	1	c_2	1	c_2	1	c_2
0	1	1	1	c_3	c_4	c_3	c_4	x
1	0	0	c_5	1	1	c_5	c_5	1
1	0	1	c_6	1	c_7	c_6	x	c_7
1	1	0	c_8	c_9	1	x	c_8	c_9
1	1	1	0	0	0	x	x	x

Table III: Codewords for 3-output functions

The algorithm which allows such identification imposes the selection of a 0 or 1 logic value for the c_i symbols in the code table. Note that the trivial solution ($F = G$) can be simply derived from the code table by imposing all c_i terms equal to 0 in the column vectors corresponding to the g_k functions ($k = 0, 1, \dots, n-1$), so that the g_k functions can be neglected.

Let us consider the column vectors g_k of the code table. In the first step, we select the first column vector (g_0) and we minimize the number of 0s in the column vector by imposing all c_i symbols equal to 1. In the second step, we select the second column vector (g_1) and we can note that the pair (g_0, g_1) satisfies the condition that the already defined elements of both columns are equal to each other. Thus, we set to 1 those c_i symbols of g_1 which correspond to ones in column g_0 . After this step the two columns g_0 and g_1 are equal to each other, then we can discard one of them, thus reducing the required redundancy. We iterate this procedure for each column vector g_k of the code table and, after each step, we update the values of c_i in all the remaining column vectors of the table. In this way we derive a single function (g_0).

Let us consider the generic column vector g_{kr} ($k = 0, 1, \dots, n-1, r = 0, 1, \dots, n-1, r \neq k$) of the code table and let us simply indicate it as g_k . In the k -th step, we select the k -th column vector (g_k) and we minimize the number of 1s in the column vector. In the $(k+1)$ -th step, we select the $(k+1)$ -th column vector (g_{k+1}) and we first verify if the pair (g_k, g_{k+1}) satisfies the condition that the already defined elements of each column are equal. We have two possibilities:

1. If the condition is verified, we try to equal g_{k+1} to g_k by imposing the corresponding c_i symbols (that is the c_i symbols having the same position in the column vector) of column g_{k+1} equal to those of column g_k . For the not corresponding c_i element of both vectors, we let the c_i symbols unset and we will consider them in a later step.
2. Otherwise, we leave the c_i symbols not assigned for later optimization.

Then, if the g_k and g_{k+1} columns will result equal to each other, then we can discard one of them, thus reducing the required redundancy. If the two columns are different, we consider the $(k+2)$ -th column with respect to the k -th and to the $(k+1)$ -th columns (pair $g_{k+2}-g_k$ and pair $g_{k+2}-g_{k+1}$). We iterate this procedure for each column vector of the code table and, after each step, we update the values of the c_i elements in all the remaining column vectors of the table.

Let us derive, as simple case study, the proposed AUED codes for both the 2- and 3-output functions. In the first case ($k = 0, 1$), we obtain ($2 \cdot 3^2 = 18$) possible codes (see Table II). Depending on the last x value on g_{01} , we have two code subclasses. Each subclass is composed of 9 codes. All the codes belonging to the first subclass ($x = 0$) have the properties that the redundant logic is unate negative.

Among codes of the first subclass ($x = 0$), by imposing both c_i equal to 0 in the column vectors corresponding to the g_0 and g_1 functions, the trivial code ($F = G$) is derived, hence the

g_{01} function can be neglected. Conversely, by setting to 1 both c_i elements in the g_0 and g_1 column vectors, the remaining c_i terms must be zeroed in the g_{01} column vector. The resulting code, called $i2$ and represented in Tab. IV, requires less redundant bits (2 additional bits instead of 3), being $g_1 = g_0$.

For 3-output functions ($k = 0, 1, 2$), the total number of codes is very large ($2^6 3^6 7^3$) (see Table III). Among the unate negative codes ($x = 0$), the trivial code ($F = G$) can be obtained or, by setting to 1 all c_i symbols in the g_0, g_1 and g_2 column vectors and setting to 0 the c_i symbols in the g_{01}, g_{02} and g_{12} columns, we can derive the code $i3$ (represented in Tab. V) imposing less redundant bits (4 additional bits instead of 6) and providing minimum unate synthesis.

The unate negative codes are best suited for implementation in Fully CMOS architectures proposed in [5] and [6], in order to obtain also fault tolerant characteristics.

f_0	f_1	g_0	g_1	g_{01}
0	0	1	1	1
0	1	1	1	0
1	0	1	1	0
1	1	0	0	0

Table IV: Codewords for code $i2$

f_0	f_1	f_2	g_0	g_{01}	g_{02}	g_{12}
0	0	0	1	1	1	1
0	0	1	1	1	0	0
0	1	0	1	0	1	0
0	1	1	1	0	0	0
1	0	0	1	0	0	1
1	0	1	1	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0

Table V: Codewords for code $i3$

III. Code Synthesis Method

A further optimization in terms of both the number of redundant functions and the area devoted to the redundant logic can be obtained by directly synthesizing the code functions g_k and g_{kr} from the primary inputs of the original circuit to be protected. Figure 1 shows the proposed self-checking architecture, where the synthesis of the functional block F and those of the code bit generator G are provided separately. Further optimization can be obtained by using a single block to synthesize both the functional outputs and the redundant functions from the primary inputs. The application of the proposed synthesis approach allows us to minimize both the number of redundant code functions and the area required by the code bit generator G by exploiting the intrinsic characteristics of the original logic to be protected.

The synthesis of the functional block F can be achieved by using a classical synthesis approach. Conversely, the proposed method is devoted to the synthesis of the redundant block, G , where a set of conditions existing among the c_i symbols of each row can be exploited during the minimization. Due to the presence of such conditions, a classical synthesis approach cannot be applied. As a matter of fact, the c_i symbols cannot be considered as usual *don't cares* (DCs) but rather *conditional don't cares* ($COND-DCs$), since they are related to each other by requiring that at least one of them in each row must be set to 0, while the remaining ones are usual DCs .

A minimal covering problem must be solved to synthesize the redundant block, G , therefore we defined an *ad hoc* heuristic algorithm aiming at reducing both the number of required redundant functions and the logic to implement the remaining redundant functions. Hereafter we briefly summarize the proposed synthesis algorithm. Since the $COND-DC$ symbols impose conditions such as at least one zero over n symbols, the synthesis method is based on the search of primary *implicates* instead of primary *implicants*, thus in the following we adopt the product of sums representation.

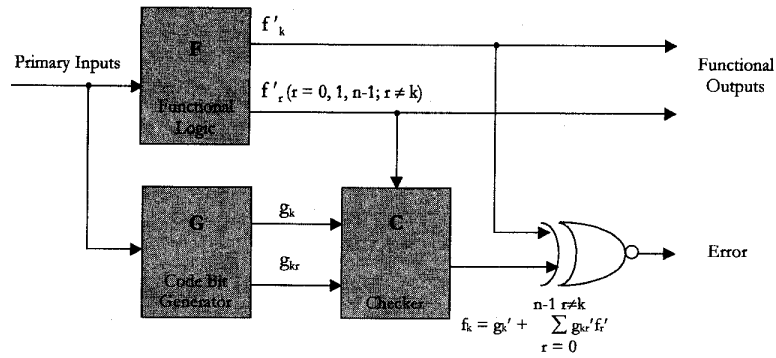


Fig. 1: Self-Checking Architecture

The proposed synthesis algorithm consists of two main parts: (a) finding all prime implicants of the redundant functions and (b) selecting a minimal set of prime implicants that cover the redundant functions. Both parts of the algorithm are based on a two-dimensional *cover table* defined as follows. First, all the *implicants* (sum terms) for each redundant function are derived by considering the *COND-DC* and x symbols as zeros. Second, the rows of the cover table correspond to the implicants (I_i) for all the redundant functions (an implicate can appear on more than one row if it corresponds to more than one redundant function). Third, the columns of the cover table represent all the different c_j symbols that are contained in the redundant functions. Finally, each entry m_{ij} of the cover table is a symbol equal to * if and only if the implicate I_i for the row i covers the c_j symbol for the column j . In the general case, some zero terms, not related to the c_j symbols, can appear in the code table. Therefore the corresponding maxterms must be added in the columns of the cover table.

As an example, let us consider the application of the 3-output AUED code shown in Table III to a 2-bit binary adder with 4 primary inputs (a_0, b_0, a_1, b_1). The logic behavior of the 2-bit adder outputs and the related code functions with respect to the primary inputs are summarized in Table VI. It should be noted that in this particular application the output configuration $f_0 = f_1 = f_2 = 1$ is not used. Therefore, in this example, without any loss of generality, the zero terms in the code table are related to the c_j terms only. The corresponding cover table is depicted in Table VII. The cover table has 25 rows corresponding to the implicants of the redundant functions and 22 columns corresponding to the c_j terms appearing in the redundant functions.

Given the cover table, the problem can be stated as to select a minimum subset of rows that cover all the columns. The aim of the proposed algorithm is twofold. From one side, it aims at reducing the total number of implicants and possibly the number of related literals, as the Quine McCluskey algorithm does. From the other side, the concept of weight associated with each redundant function has been introduced to reduce the number of required redundant functions, trying to eliminate the redundant functions with higher cost in terms of the ratio among number of required literals and c_j terms covered.

(a) Finding all prime implicants

Due to the definition of *COND-DCs*, the cover table does not contain essential prime implicants related to the c_j symbols. However, essential prime implicants can appear in the cover table due to zero terms in the code table. In this case, the essential prime implicants must be included in the minimal cover.

a_i	b_i	a_0	b_0	f_0	f_1	f_2	g_0	g_1	g_2	g_{01}	g_{02}	g_{12}	
0	0	0	0	0	0	0	1	1	1	1	1	1	
0	0	0	1	0	0	1	1	1	c_1	1	c_1	c_1	
0	0	1	0	0	0	1	1	1	c_2	1	c_2	c_2	
0	0	1	1	0	1	0	1	c_3	1	c_3	1	c_3	
0	1	0	0	0	1	0	1	c_4	1	c_4	1	c_4	
0	1	0	1	0	1	1	1	c_5	c_6	c_5	c_6	X	
0	1	1	0	0	1	1	1	c_7	c_8	c_7	c_8	X	
0	1	1	1	1	0	0	c_9	1	1	c_9	c_9	1	
1	0	0	0	0	1	0	1	c_{10}	1	c_{10}	1	c_{10}	
1	0	0	1	0	1	1	1	c_{11}	c_{12}	c_{11}	c_{12}	X	
1	0	1	0	0	1	1	1	c_{13}	c_{14}	c_{13}	c_{14}	X	
1	0	1	1	1	1	0	0	c_{15}	1	1	c_{15}	c_{15}	1
1	1	0	0	1	0	0	c_{16}	1	1	c_{16}	c_{16}	1	
1	1	0	1	1	0	1	c_{17}	1	c_{18}	c_{17}	X	c_{18}	
1	1	1	0	1	0	1	c_{19}	1	c_{20}	c_{19}	X	c_{20}	
1	1	1	1	1	1	0	c_{21}	c_{22}	1	X	c_{21}	c_{22}	

Table VI: Output functions and code functions with respect to the primary inputs for the 2-bit adder.

The rows of the cover table corresponding to each essential prime implicate and the corresponding *covered* columns (the columns with a * symbol for each essential prime implicate) must be eliminated from the cover table. Then, this part of the algorithm consists of the following steps:

- 1) Assign a *weight* w_g to each redundant function corresponding to the number of literals appearing in all the implicates of the function divided by the number of the * symbols appearing in the function. In the example we have $w_{g_0} = 1$, $w_{g_1} = 2$, $w_{g_2} = 0.5$, $w_{g_{01}} = 0.24$, $w_{g_{02}} = 0.71$, $w_{g_{12}} = 1.6$.
- 2) Order the redundant functions by w_g in decreasing order: $[g_1, g_{12}, g_0, g_{02}, g_2, g_{01}]$
- 3) Starting from the beginning of the list (g_j) compare each row r_i of the selected function with all rows of the other functions that are below in the list.
- 4) A row (r_i) of the cover table is said to be *covered* (or dominated) by a row r_j if all its * symbols are already present in the same position in the row r_j that belongs to a redundant function g_j corresponding to a weight w_{g_j} such that $w_{g_j} \leq w_{g_i}$. If a row in the selected redundant function is found to be covered by another row, then delete the covered row from the cover table.

It should be noted that the c_i terms with the same index i correspond to the same row in the original code table (see for example Table VI), therefore, by construction, they correspond to the same maxterm. Moreover during the synthesis steps it is mandatory to maintain at least one * symbol for each column c_i of the cover table.

- 5) Repeat this step until each implicate in the table has been compared with all the remaining ones below it. In particular, if all the rows of a redundant function have been found to be covered by rows of other redundant functions, then the redundant function can be eliminated from the cover table.
- 6) The remaining rows in the cover table are the prime implicates of the redundant function. In our example, the dominated rows are the shaded rows in Table VII (rows 2, 3, 4, 5, 6, 7, 8, 9, 16, 17, 20, 21, 24, 25), while Table VIII reports the prime implicates cover table, where the dominated rows have been eliminated.

	Implicates	F	c ₁	c ₂	c ₃	c ₄	c ₅	c ₆	c ₇	c ₈	c ₉	c	c	c	c	c	c	c	c	c	c	c	c	c
1	a ₁ '+b ₁ '	g ₀																*	*		*	*		
2	a ₀ '+b ₀ '+b ₁ '	g ₀								*													*	*
3	a ₀ '+b ₀ '+a ₁ '	g ₀													*								*	*
4	a ₀ +a ₁ +b ₁ '	g ₁				*	*																	
5	b ₀ +a ₁ +b ₁ '	g ₁				*		*																
6	a ₀ +a ₁ '+b ₁	g ₁								*	*													
7	b ₀ +a ₁ '+b ₁	g ₁								*	*		*											
8	a ₀ '+b ₀ '+a ₁ '+b ₁ '	g ₁																						*
9	a ₀ '+b ₀ '+a ₁ +b ₁	g ₁			*																			
10	a ₀ +b ₀ '	g ₂	*				*					*								*				
11	a ₀ '+b ₀	g ₂	*					*					*							*				*
12	b ₁ '	g ₀₁			*	*		*	*						*	*	*	*	*	*	*	*	*	*
13	a ₁ '	g ₀₁			*	*		*	*		*	*	*	*	*	*	*	*	*	*	*	*	*	*
14	a ₀ '+b ₀ '	g ₀₁		*					*					*				*		*		*		*
15	a ₁ '+b ₁ '	g ₀₂								*					*			*		*		*		*
16	a ₀ +b ₀ '	g ₀₂	*				*				*			*			*		*		*		*	*
17	a ₀ '+b ₀	g ₀₂	*		*			*	*			*		*		*		*		*		*		*
18	a ₀ '+b ₁ '	g ₀₂						*	*						*	*		*		*		*		*
19	a ₀ '+a ₁ '	g ₀₂								*	*			*	*		*		*		*		*	*
20	a ₀ +b ₀ '	g ₁₂	*														*		*		*		*	*
21	a ₀ '+b ₀	g ₁₂	*		*												*		*		*		*	*
22	a ₀ '+a ₁ '+b ₁ '	g ₁₂	*		*												*		*		*		*	*
23	b ₀ '+a ₁ +b ₁	g ₁₂	*		*												*		*		*		*	*
24	a ₀ +a ₁ '+b ₁	g ₁₂								*							*		*		*		*	*
25	a ₀ +a ₁ +b ₁ '	g ₁₂			*						*						*		*		*		*	*

Table VII: Cover table for the 2-bit adder.

	Prime Implicates	F	c ₁	c ₂	c ₃	c ₄	c ₅	c ₆	c ₇	c ₈	c ₉	c	c	c	c	c	c	c	c	c	c	c	c	c
1	a ₁ '+b ₁ '	g ₀															*	*		*	*			
10	a ₀ +b ₀ '	g ₂	*				*					*							*					
11	a ₀ '+b ₀	g ₂	*		*			*	*				*						*		*		*	*
12	b ₁ '	g ₀₁			*	*		*	*						*	*	*	*	*	*	*	*	*	*
13	a ₁ '	g ₀₁			*	*		*	*		*	*	*	*	*	*	*	*	*	*	*	*	*	*
14	a ₀ '+b ₀ '	g ₀₁		*					*					*			*		*		*		*	*
15	a ₁ '+b ₁ '	g ₀₂								*				*			*		*		*		*	*
18	a ₀ '+b ₁ '	g ₀₂						*	*					*	*		*		*		*		*	*
19	a ₀ '+a ₁ '	g ₀₂								*	*			*	*		*		*		*		*	*
22	a ₀ '+a ₁ '+b ₁ '	g ₁₂	*		*												*		*		*		*	*
23	b ₀ '+a ₁ +b ₁	g ₁₂	*		*												*		*		*		*	*

Table VIII: Prime implicates cover table for the 2-bit adder showing the distinguished 1-cells and the essential prime implicates as shaded

(b) Minimizing the cover

Once the list of all prime implicates has been derived, the minimization step is based on the selection of a minimal subset of them aiming at both covering all the 0's of the redundant functions and minimizing the number of 0's due to the COND-DCs. Starting from the prime implicates cover table (Table VIII), this part of the algorithm consists of the following steps:

- 1) Identify the distinguished I -cells as the table columns with a single I , as shown by the shaded columns in Table VIII.
- 2) A row that contains a * symbol in one or more distinguished 1-cell columns corresponds to an *essential prime implicate*. Include all essential prime implicates in the minimal cover. In our example, the implicates (10, 11, 12, 13 and 22) are essential prime implicates.
- 3) Remove the essential prime implicates and the columns they cover (the shaded rows and columns of Table VIII plus the *covered columns* 1, 8, 9, 14, 15, 16, 17, 18 and 19). If any row is empty, it is deleted; the corresponding prime implicates are redundant, that is completely covered by essential prime implicates. The reduced cover table for the selected example is shown in Table IX. In this step column dominance is also analyzed. A column c_i that contains * symbols at least in the same positions of column c_j is said a *dominant column* and can be deleted from the cover table. In the application example no column dominance occurs.
- 4) Remove any prime implicates that are eclipsed by others with equal or less cost or to reduce the number of redundant functions. In the reduced cover table this is done by deleting any rows whose checked columns are a proper subset of another row's and deleting all but one of a set of rows with identical columns. Table IX shows in colour the eclipsed rows, while Table X shows the resulting table after removal of the eclipsed rows.
- 5) Identify the distinguished 1-cells (Table X) and includes the corresponding *secondary essential prime implicates* in the minimal cover. As before, any row that contains a * symbol in one or more distinguished 1-cell columns corresponds to a secondary essential prime implicate (14 and 18).
- 6) If all remaining columns are covered by the secondary essential prime implicates as in Table X, then the algorithm stops. Otherwise if any secondary essential prime implicates were found in the previous step, we go back to step 4) and we iterate.
- 7) Otherwise a branch and bound algorithm must be used. This implies selecting rows one at a time and treating them as if they are essential, and recursing steps from 3 to 6.

	<i>Prime Implicates</i>	F	c_1	c_2
1	$a_1'+b_1'$	g_0	*	*
14	$a_0'+b_0'$	g_{01}	*	*
15	$a_1'+b_1'$	g_{02}	*	*
18	$a_0'+b_1'$	g_{02}	*	*
19	$a_0'+a_1'$	g_{02}	*	*
23	$b_0'+a_1'+b_1$	g_{12}	*	*

Table IX: Reduced cover table after removal of essential prime implicates and the columns that they cover. The eclipsed rows are shaded

	<i>Prime Implicates</i>	F	c_1	c_2
14	$a_0'+b_0'$	g_{01}	*	*
18	$a_0'+b_1'$	g_{02}	*	*

Table X: Reduced cover table after removal of eclipsed rows, thus showing secondary essential prime implicates.

The resulting solution is depicted in Table XI. Note that the number of redundant functions (4) is equal to those required by the code $i3$, but it is double with respect to the redundancy imposed by the 3-output Berger code. In fact the Berger codes are optimal systematic AUED codes, in the sense that the information rate is the highest possible for any information block length [1]. However, the proposed code is better than the Berger code in terms of total area cost for the block G : in the case of the application example, the proposed code requires 13 literals, while the Berger code requires 33 literals.

a_1	b_1	a_0	b_0	f_0	f_1	f_2	g_2	g_{01}	g_{02}	g_{12}
0	0	0	0	0	0	0	1	1	1	1
0	0	0	1	0	0	1	0	1	1	1
0	0	1	0	0	0	1	0	1	1	1
0	0	1	1	0	1	0	1	0	1	1
0	1	0	0	0	1	0	1	0	1	1
0	1	0	1	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0	0	0	1
0	1	1	1	1	1	0	0	1	0	1
1	0	0	0	0	1	0	1	0	1	1
1	0	0	1	0	1	1	0	0	1	1
1	0	1	0	0	1	1	0	0	1	1
1	0	1	1	1	0	0	1	0	1	1
1	1	0	0	1	0	0	1	0	1	1
1	1	0	1	1	0	1	0	0	1	1
1	1	1	0	1	0	1	0	0	0	0
1	1	1	1	1	1	0	1	0	0	0

Table XI: Results of the synthesis method applied to the 2-bit adder

IV. Conclusions

We defined a class of systematic AUED codes suitable for self-checking combinational architectures. Besides, a synthesis method has been proposed to derive the redundant functions directly from the primary inputs of the original circuit, thus reducing speed penalty and area overhead, given a target self-checking architecture.

Future directions of the work are devoted to identify synthesis algorithm to map the proposed AUED codes to the fault tolerant fully CMOS architecture defined in [5, 6]. In this way we can achieve not only the detection of all unidirectional errors, but also the correction of all possible single transistor stuck-on faults.

V. References

- [1] T. Rao, E. Fujiwara, *Error Control Coding for Computer Systems*, Englewood Cliffs, NJ, Prentice Hall, 1989.
- [2] N.K. Jha, *Separable Codes for Detecting Unidirectional Errors*, IEEE Transactions on Computer-Aided Design, Vol.8, No.5, May 89, pp. 571-574.
- [3] N.K. Jha, S.-Y. Wang, *Design and Synthesis of Self-Checking VLSI Circuits*, IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 12, No. 6, June 93, pp.878-887.
- [4] S. Kundu, S.M: Reddy, *Embedded Totally Self-Checking Checkers: a Practical Design*, IEEE Design and Test of Computers, August 1990, pp.5-12.
- [5] C. Bolchini, G. Buonanno, D. Sciuto, R. Stefanelli, *Fault Detection and Fault Tolerance Issues at CMOS Level through AUED Encoding*, DFT96: 1996 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 1996.
- [6] C. Bolchini, G. Buonanno, D. Sciuto, R. Stefanelli, *CMOS Fault Tolerant Architectures for Switch Level Faults*, Journal of Microelectronics System Integration, Vol. 3, N. 2, pp.121-139, 1995.
- [7] D. Sciuto, C. Silvano, R. Stefanelli, *Systematic AUED Codes for Self-Checking Architectures*, Technical Report, Università di Brescia, Brescia, Italy, June 1998.