Reed-Muller adders based on binary stored carry-or-borrow representation

Daniel Torno, Member, IEEE Computer Society
Exorand Technology, http://www.exorand.com
Orléans, France

***Abstract*—This paper introduces new Reed-Muller implementations of Carry-Look-Ahead Adder, Ripple-Carry Adder, Half-Adder and Full-Adder. First from the standard equations used for the carry generation we derived an expression with a new generate signal. Then it is shown that the used underlying redundant representation is the binary stored carry-or-borrow representation. Based upon this representation several Reed-Muller adder implementations are then exposed with their testability results. Finally some test measures of a CMOS prototype circuit are discussed.**

I. INTRODUCTION

Over the last decades a lot of different implementations of adders have been introduced providing various performances in speed, consumption or testability, but for all these implementations the underlying number representation is the binary redundant carry-save number system. We introduce a range of binary adders based on a binary stored-carry-or-borrow representation which main characteristic is to have a rather simple Reed-Muller implementation.

First this paper introduces a new way to represent the carry process of binary addition. Starting from the basic equations of a standard Carry-Look-Ahead adder, some Boolean manipulations lead to a new recursive expression of the carry. Sum bits are expressed in EXOR and Not-EXOR terms of the input bits. Based on these equations new Reed-Muller implementations of carry-look-ahead and ripple-carry adders are proposed. These implementations are not described in the literature (see [1], [2], [3], [4], [5], [6], [7]) nor in the patent databases.

Then it is shown that the sum equations are composed of a combination of an estimation stage followed by propagation stages. These stages compute the sum by using a binary redundant representation based on a partial sum and a "carry" which can have positive and negative effects on the partial sum and identified as the binary stored carry-or-borrow representation.

A full-adder structure based on the new binary redundant representation is exposed and as an example a new Reed-Muller implementation of a three-operands addition is proposed. Finally we give some test results (area, power consumption and speed) of a CMOS circuit implementing a four bits Reed-Muller carry-look-ahead adder.

II. REED-MULLER EXPRESSION OF THE SUM BITS OF THE BINARY ADDER

A. Expression of the sum bits in Reed-Muller form

Let $a_{N-1}, a_{N-2} \cdots a_0$ and $b_{n-1}, b_{n-2} \cdots b_0$ be two N-bit binary numbers and their sum be $s_N, s_{N-1} \cdots s_0$. The input carry to the least significant bit is cy_{in}.

The Boolean expressions for partial sum s_n and carry cy_n for a single-bit adder are as follows:

$$\begin{cases} cy_n = g_{n-1} + p_{n-1} \cdot cy_{n-1} \\ s_n = p_n \oplus cy_n \end{cases} \quad (1.)$$

with: $\begin{cases} g_{n-1} = a_{n-1} \cdot b_{n-1} \\ p_{n-1} = a_{n-1} \oplus b_{n-1} \end{cases}$ for generate and propagate signals.

The carry cy_n and the generate signal g_{n-1} can also be expressed as:

$$cy_n = g_{n-1} \oplus p_{n-1} \cdot cy_{n-1}$$

This work was supported in part by a grant from OSEO Centre, France.

and $g_{n-1} = a_{n-1} \oplus \overline{b_{n-1}} \cdot p_{n-1}$.

The substitution of g_{n-1} in the expression of cy_n gives:

$$cy_n = a_{n-1} \oplus p_{n-1} \cdot \left(\overline{b_{n-1}} \oplus cy_{n-1}\right) \quad (2.)$$

We introduce a term q_n defined as follows:

$$q_n = \overline{b_n \oplus a_{n-1}} \quad , \quad q_0 = \overline{b_0 \oplus cy_{in}}$$

By developing cy_{n-1} in

$$cy_n = a_{n-1} \oplus p_{n-1} \cdot \left(\overline{b_{n-1}} \oplus cy_{n-1}\right) \quad (3.)$$

we get :

$$cy_n = a_{n-1} \oplus p_{n-1} \cdot q_{n-1} \oplus p_{n-1} \cdot p_{n-2} \cdot \left(\overline{b_{n-2}} \oplus cy_{n-2}\right)$$

And by developing until:

$$cy_0 = a_0 \oplus p_0 \cdot \left(\overline{b_{n-1}} \oplus cy_{n-1}\right) = a_0 \oplus p_0 \cdot q_0$$

we obtain:

$$cy_n = a_{n-1} \oplus p_{n-1} \cdot q_{n-1} \oplus p_{n-1} \cdot p_{n-2} \cdot q_{n-2} \oplus \ldots\ldots \oplus p_{n-1} \cdot p_{n-2} \cdots\cdots p_1 \cdot q_1 \oplus p_{n-1} \cdot p_{n-2} \cdots\cdots p_1 \cdot p_0 \cdot q_0 \quad (4.)$$

We obtain an equation for cy_n expressed in terms of the propagate signal p_{n-j} and a new generate signal q_{n-j}.

For the sake of memory, when using EXOR gates the conventional expression of cy_n is:

$$cy_n = g_{n-1} \oplus p_{n-1} \cdot g_{n-2} \oplus p_{n-1} \cdot p_{n-2} \cdot g_{n-3} \oplus \ldots \oplus p_{n-1} \cdot p_{n-2} \cdots p_1 \cdot g_0 \oplus p_{n-1} \cdot p_{n-2} \cdots p_1 \cdot p_0 \cdot c_{in} \quad (5.)$$

Or

$$cy_n = g_{n-1} + p_{n-1} \cdot g_{n-2} + p_{n-1} \cdot p_{n-2} \cdot g_{n-3} + \ldots + p_{n-1} \cdot p_{n-2} \cdots p_1 \cdot g_0 + p_{n-1} \cdot p_{n-2} \cdots p_1 \cdot p_0 \cdot c_{in} \quad (6.)$$

Both expressions (4) and (5) are very similar, they can be implemented with the same amount of gates. But in the expression (4), OR operators can not be substituted to the EXOR as it is possible in the conventional expression (5). Minterms of equation (4) are not disjoint as these of equation (5), for instance:

$$p_{n-1} \cdot q_{n-1} \cdot p_{n-2} \cdot q_{n-2} \neq 0$$

$$p_{n-1} \cdot g_{n-2} \cdot p_{n-2} \cdot g_{n-3} = 0$$

B. Carry-look-ahead adder implementation

For a four bits implementation, let be cy_{in} the input carry and cy_{out} the output carry.

With following terms:

$$q_0 = \overline{b_0 \oplus cy_{in}},$$

$$q_n = \overline{b_n \oplus a_{n-1}},$$

$$p_n = a_n \oplus b_n$$

A 4 bit adder can be expressed by the set of equations (7) implemented in fig. 1.

$$\begin{cases} s_0 = p_0 \oplus cy_{in} \\ s_1 = \overline{a_1} \oplus q_1 \oplus p_0 \cdot q_0 \\ s_2 = \overline{a_2} \oplus q_2 \oplus p_1 \cdot q_1 \oplus p_1 \cdot p_0 \cdot q_0 \\ s_3 = \overline{a_3} \oplus q_3 \oplus p_2 \cdot q_2 \oplus p_2 \cdot p_1 \cdot q_1 \oplus p_2 \cdot p_1 \cdot p_0 \cdot q_0 \\ cy_{out} = a_3 \oplus p_3 \cdot q_3 \oplus p_3 \cdot p_2 \cdot q_2 \oplus p_3 \cdot p_2 \cdot p_1 \cdot q_1 \oplus p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot q_0 \end{cases} \quad (7.)$$

Structure displayed in following fig. 1 is very similar to these of the known carry-look-ahead adder and this four bits implementation can be easily extended to larger data width in the same ways as standard Carry-Look-Ahead adders with group propagate and generate signals.

Testability of adders implemented with EXOR gates has been studied in [8].

The proposed implementation has the same number of gates than the 4 bits Carry-Look-Ahead adder using AND, EXOR gates described in [8] but used types are different:

- 22 EXOR gates, 10 AND gates versus 18 EXOR gates, 14 AND gates for the proposed implementation.

To find a minimum test vectors set, first Tétramax tool from Synopsis was used to generate all possible stuck-at-faults.

Then a heuristic software named Setcovergin and developed at LIRMM [9] has computed a minimal test coverage.

The testability of the proposed implementation is improved over the standard implementation as a set of 8 test vectors is sufficient to detect all single stuck-at faults. This is due to the first stage composed uniquely of EXOR gates.

For larger data width it should be interesting to look if a "log(n) testability" formula exists as it was demonstrated for standard carry-look-ahead implementation in [10].

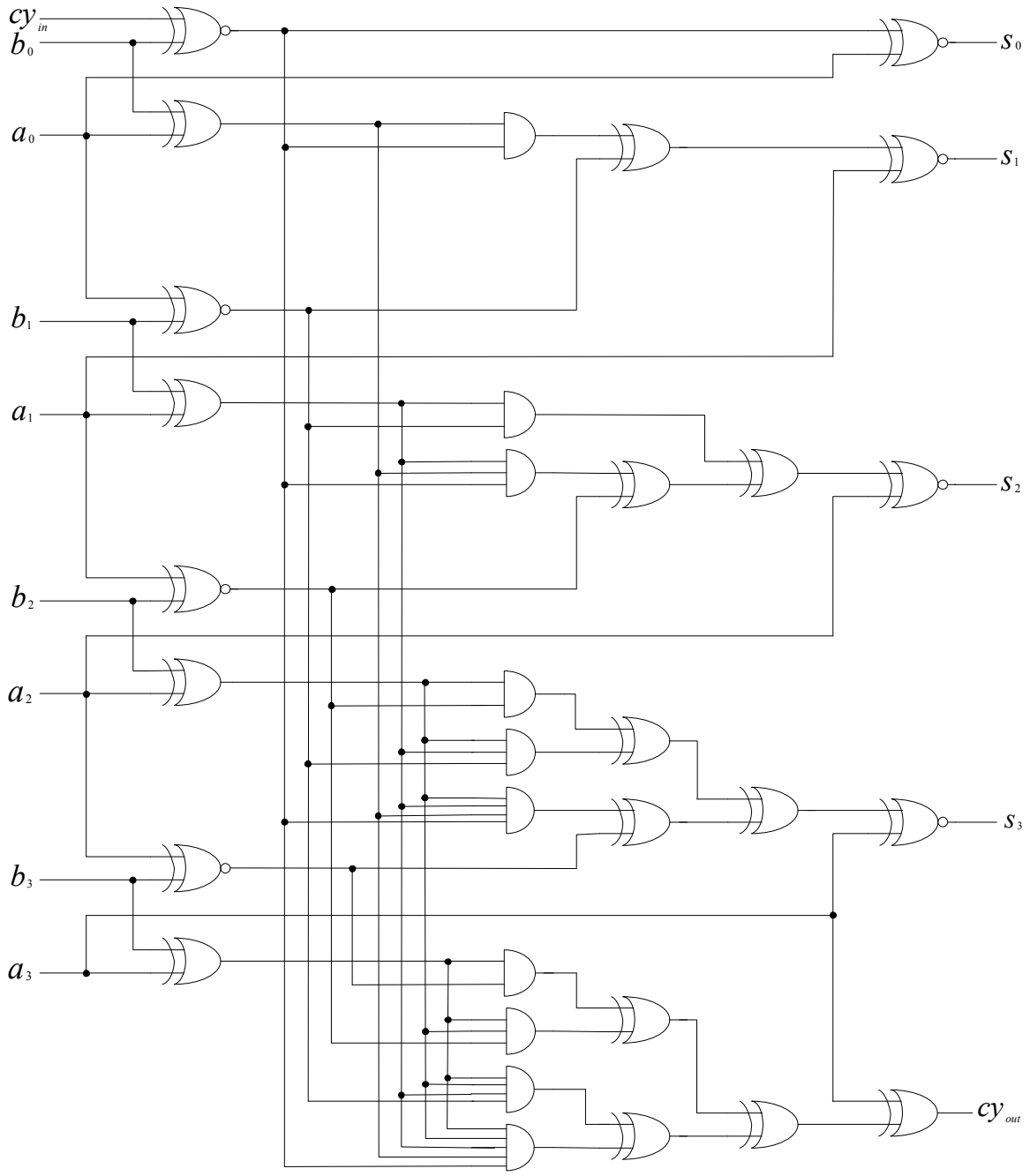

Figure 1. Four bits carry-look-ahead adder

C. Ripple-carry adder implementation

For a ripple-carry adder composed of identical cells, we introduce z_i, a signal propagated from the cell (n) to the cell (n+1).

Then the equations for a ripple-carry implementation can be deduced from the equations (7).

$$\begin{cases} s_n = \overline{a_n} \oplus q_n \oplus z_{n-1} \\ z_n = p_n \cdot \left[q_n \oplus z_{n-1} \right] \end{cases} \quad (8.)$$

with: $\begin{cases} q_n = \overline{b_n \oplus a_{n-1}} \\ p_n = a_n \oplus b_n \end{cases}$

Figure 2 shows a 4 bits Ripple-Cary adder organized as one bit functional cells.

This implementation is C-testable and a set of 4 test vectors is sufficient to detect all single stuck-at faults.

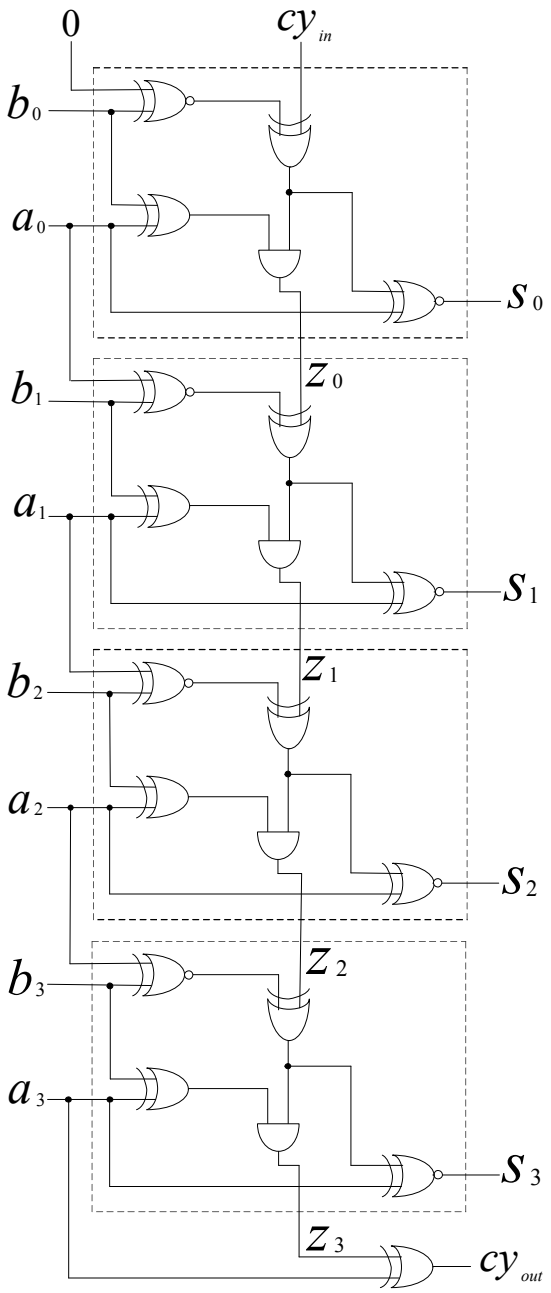

Figure 2. Four bits ripple-carry adder

III. BINARY STORED CARRY-OR-BORROW REPRESENTATION

A. Sum expressed as estimation and propagation

It is well known that an addition of two binary numbers A and B can be realized by a network of half-adders organized in a half-pyramid structure (figure 3).

Each half-adder computing carry/save bits according to followings equations for the first stage:

$$\begin{cases} S_n^0 = a_n \oplus b_n \\ CY_{n+1}^0 = a_n \cdot b_n \end{cases} \quad (9.)$$

and for the next stages to equations (10).

$$\begin{cases} S_n^i = S_n^{i-1} \oplus CY_n^{i-1} \\ CY_{n+1}^i = S_n^{i-1} \cdot CY_n^{i-1} \end{cases} \quad (10.)$$

In fact the adding structure is the same for the first stage and next stages as shown fig. 3.

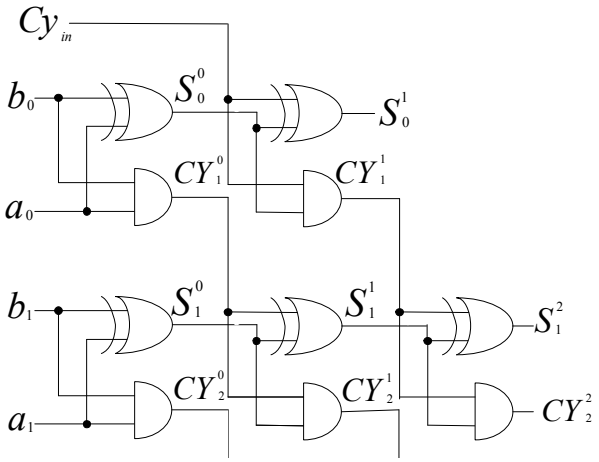

Figure 3. Addition process with a carry/save half-adder network

By an empirical way the author has found a new expression of this adding process: this new expression is composed by two different computation stages.

The first one computes an initial estimation of the sum expressed by the two terms U_n^0 and R_{n+1}^0:

$$\text{Initial estimation: } \begin{cases} U_n^0 = \overline{a_n} \\ R_{n+1}^0 = \overline{a_n \oplus b_{n+1}} \end{cases} \quad (11.)$$

The second one computes partial result U_n^0 and propagates the « carry » R_{n+1}^0:

$$\text{Propagation equations: } \begin{cases} U_n^i = U_n^{i-1} \oplus R_n^{i-1} \\ R_{n+1}^i = \left(U_n^{i-1} \oplus U_{n-1}^{i-1}\right) \cdot R_n^{i-1} \end{cases} \quad (12.)$$

Figure 4 shows a four bits implementation of a network realized with these two stages.

The variable R_{n+1}^i could have positive or negative effects on partial sum U_{n+1}^i.

Table I displays the intermediate values of an addition process (A = 00100, B = 00111 , input carry cy_{in} = 1). The initial approximation has the value 11011 and after propagation of the carry-borrow the value of the final result is 01100.

The partial result expressed as U_n^i and R_{n+1}^i is a binary redundant expression which can be identified as binary stored-carry-or-borrow (noted as BSCB) representation (see [3]).

TABLE I. TABLE TYPE STYLES

Values	Position					
	4	**3**	**2**	**1**	**0**	cy_{in}
A	*0*	*0*	*1*	*0*	*0*	*1*
B	*0*	*0*	*1*	*1*	*1*	*0*
$U_n^0 = \overline{a_n}$	***1***	***1***	***0***	***1***	***1***	*0*
$R_n^0 = \overline{b_n \oplus a_{n-1}}$	*1*	*0*	*0*	*0*	*1*	
$U_n^1 = U_n^0 \oplus R_n^0$	*0*	*1*	*0*	*1*	*0*	
$R_n^1 = \left(U_{n-1}^0 \oplus U_{n-2}^0\right).R_{n-1}^0$	*0*	*0*	*0*	*1*		
$U_n^2 = U_n^1 \oplus R_n^1$	*0*	*1*	*0*	*0*	*0*	
$R_n^2 = \left(U_{n-1}^1 \oplus U_{n-2}^1\right).R_{n-1}^1$	*0*	*0*	*1*			
$U_n^3 = U_n^2 \oplus R_n^2$	*0*	*1*	*1*	*0*	*0*	
$R_n^3 = \left(U_{n-1}^2 \oplus U_{n-2}^2\right).R_{n-1}^2$	*0*	*0*				
$U_n^4 = U_n^3 \oplus R_n^3$	*0*	*1*	*1*	*0*	*0*	
$R_n^4 = \left(U_{n-1}^3 \oplus U_{n-2}^3\right).R_{n-1}^3$	*0*					
S	***0***	***1***	***1***	***0***	***0***	

As U_1^1 and U_0^1 have opposite values the carry-or-borrow term R_1^1 is propagated to R_2^2 (cells in gray).

B. Comparison of binary redondant representation

Following tables show how carry-save and borrow-save binary redundant representations expressed the binary numbers compared to BSCB representation.

TABLE II. CARRY-SAVE REPRESENTATION

cy_{n+1}	s_n	Sum_n
0	0	0
0	1	+1
1	0	+2
1	1	+3

$0 \le Sum_n \le +3$ and $Sum_n = 2 \times cy_{n+1} + s_n$

TABLE III. BORROW-SAVE REPRESENTATION

bw_{n+1}	s_n	Sum_n
0	0	0
0	1	+1
1	0	-2
1	1	-1

$-2 \le Sum_n \le +1$ and $Sum_n = -2 \times bw_{n+1} + s_n$

TABLE IV. BSCB REPRESENTATION

R_{n+1}	U_n	Sum_n
0	0	0
0	1	+1
1	0	+2
1	1	-1

$-1 \le Sum_n \le +2$

It should be noted that for the U/R BSCB representation there is no arithmetic relationship between the binary representation and the sum value as for carry-save and borrow-save representations.

The BSCB representation expresses the addition result in a range which is an intermediate value between ranges of carry-save and borrow-save representations.

C. Use of different initial estimations

It is demonstrated in [11] that by developing the propagation equations (10) we get:

$$\begin{cases} U_n^i = U_n^0 \oplus R_n^0 \oplus R_n^1 \oplus \ldots \oplus R_n^{i-2} \oplus R_n^{i-1} \\ R_{n+1}^i = \left(U_n^0 \oplus R_n^0 \oplus U_{n-1}^0 \oplus 1\right) \cdot \left(U_{n-1}^0 \oplus R_{n-1}^0 \oplus U_{n-2}^0 \oplus 1\right) \ldots \\ \qquad \ldots \left(U_{n+2-i}^0 \oplus R_{n+2-i}^0 \oplus U_{n+1-i}^0 \oplus 1\right) \cdot R_{n+1-i}^0 \end{cases} \quad (13.)$$

Starting from the above BSCB equations (12) and considering that these equations handles binary numbers within the range -1 to +2, we will apply three different initial estimations thus generating different implementations.

By taking following values for the initial estimation:

1. At 0: $\begin{cases} U_0^0 = \overline{a_0} \\ R_0^0 = \overline{cy_{in} \oplus b_0} \end{cases}$ at n: $\begin{cases} U_n^0 = \overline{a_n} \\ R_{n+1}^0 = \overline{a_n \oplus b_{n+1}} \end{cases}$

$-1 \le Range\,of\,the\,initial\,estimation \le +2$

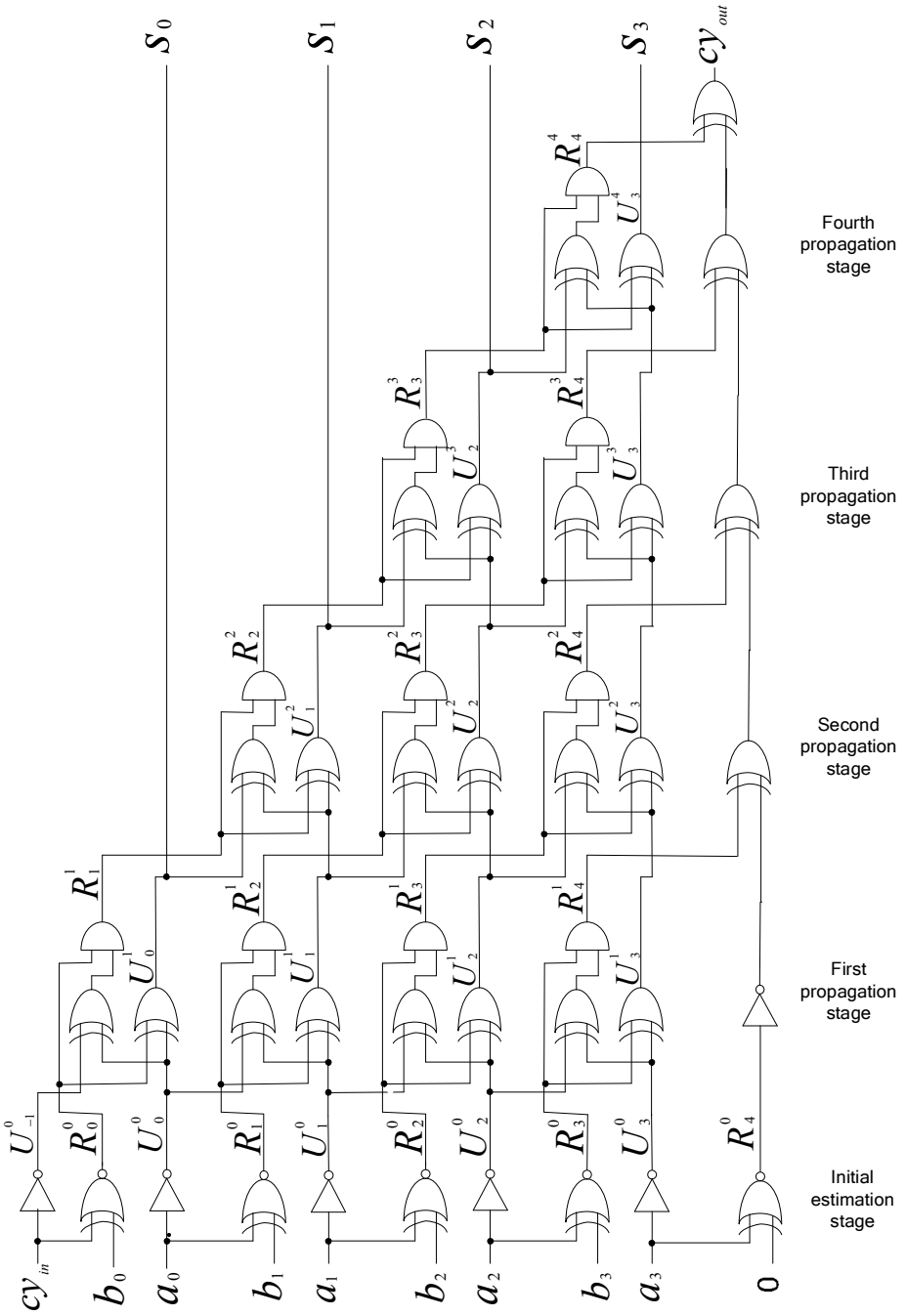

Figure 4. BSCB half-adder network

By computing R_{n+1}^{i} values in equations (13), we get (14).

$$\begin{aligned} S_3 = U_3^4 = \overline{a_3} \oplus \left(\overline{a_2 \oplus b_3}\right) \oplus (a_2 \oplus b_2)\cdot\left(\overline{a_1 \oplus b_2}\right) \\ \oplus (a_2 \oplus b_2)\cdot(a_1 \oplus b_1)\cdot\left(\overline{a_0 \oplus b_1}\right) \\ \oplus (a_2 \oplus b_2)\cdot(a_1 \oplus b_1)\cdot(a_0 \oplus b_0)\cdot\left(\overline{cy_{in} \oplus b_0}\right) \end{aligned} \tag{14.}$$

This leads to the implementation of fig 1.

2. At 0: $\begin{cases} U_0^0 = a_0 \oplus b_0 \\ R_0^0 = cy_{in} \end{cases}$ at n: $\begin{cases} U_n^0 = a_n \oplus b_n \\ R_{n+1}^0 = a_n \cdot b_n \end{cases}$

$0 \le Range\ of\ the\ initial\ estimation \le +2$

$$\begin{aligned} S_3 = U_3^4 = (a_3 \oplus b_3) \oplus (a_2 \cdot b_2) \oplus (a_2 \oplus b_2)\cdot(a_1 \cdot b_1) \\ \oplus (a_2 \oplus b_2)\cdot(a_1 \oplus b_1)\cdot(a_0 \cdot b_0) \\ \oplus (a_2 \oplus b_2)\cdot(a_1 \oplus b_1)\cdot(a_0 \oplus b_0)\cdot cy_{in} \end{aligned} \tag{15.}$$

This is the equation (5) of the standard carry-look-ahead form.

3. At 0: $\begin{cases} U_0^0 = \overline{a_0 \oplus b_0} \\ R_0^0 = cy_{in} \end{cases}$ at n: $\begin{cases} U_1^0 = \overline{a_n \oplus b_n} \\ R_{n+1}^0 = \overline{a_n} \cdot \overline{b_n} \end{cases}$

$-1 \le Range\ of\ the\ initial\ estimation \le +1$

$$S_3 = U_3^4 = (a_3 \oplus b_3) \oplus (a_2 . b_2) \oplus (a_2 \oplus b_2) . (a_1 . b_1) \oplus (a_2 \oplus b_2) . (a_1 \oplus b_1) . (a_0 . b_0) \oplus (a_2 \oplus b_2) . (a_1 \oplus b_1) . (a_0 \oplus b_0) . cy_{in}$$

(16.)

This initial approximation gives the same result as case 2.

D. Multi-operands addition

Let $a_{N-1} \cdots a_n \cdots a_1, a_0$, $b_{N-1} \cdots b_n \cdots b_1, b_0$ and $c_{N-1} \cdots c_n \cdots c_1, c_0$ be three N-bit binary numbers and their sum be: $s_{N+1}, s_N, s_{N-1} \cdots s_0$.

$$\begin{cases} A = a_{N-1}.2^{N-1} + ... + a_n.2^n + ... + a_1.2^1 + a_0.2^0 \\ B = b_{N-1}.2^{N-1} + ... + b_n.2^n + ... + b_1.2^1 + b_0.2^0 \\ C = c_{N-1}.2^{N-1} + ... + c_n.2^n + ... + c_1.2^1 + c_0.2^0 \end{cases}$$

A BSCB expression of a three operands sum can easily be deduced from the carry/save expression.

The Boolean expressions for partial sum s_n and carry cy_{n+1} for a traditional full-adder are:

$$\begin{cases} s_n = a_n \oplus b_n \oplus c_n \\ cy_{n+1} = a_n . b_n \oplus c_n . a_n \oplus c_n . b_n \end{cases} \quad (17.)$$

By considering that s_n and cy_{n+1} are two binary numbers to be added, we compute a first approximation of the sum of these two numbers according to initial approximation of case 1. We get:

$$\begin{cases} U_n = \overline{s_n} \\ R_{n+1} = \overline{cy_{n+1} \oplus s_n} \end{cases} \quad (18.)$$

By substituting s_n and cy_{n+1} values, and after some boolean manipulations we obtain:

$$\begin{cases} U_n = \overline{a_n \oplus b_n \oplus c_n} \\ R_{n+1} = (\overline{a_n \oplus b_n}) . (\overline{b_n \oplus c_n}) \end{cases} \quad (19.)$$

A rather simple implementation is given below (fig. 5). This structure is equivalent to the full-adder but uses the BSCB representation.

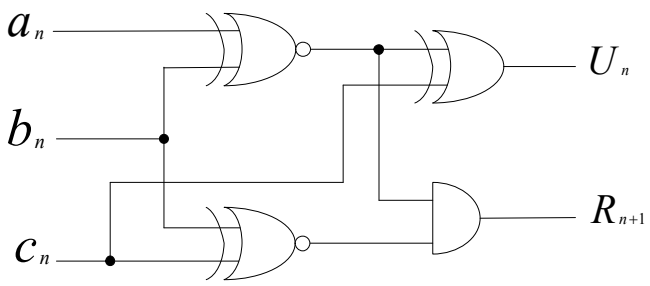

Figure 5. BSCB full-adder

By combining these BSCB full-adder to the previous described ripple-carry adder we obtain the three operands implementation displayed fig. 7.

As U_n and R_{n+1} values can not be considered as two binary numbers with no relationship (as s_n and cy_{n+1} for the carry/save representation) the previous implementation don't apply and it is necessary to think about a different way to accumulate several binary numbers.

One may build an initial BSCB expression U_n^0 and R_{n+1}^0 from two binary numbers A and B. Let be:

$$\begin{cases} U_n^0 = \overline{a_n} \\ R_{n+1}^0 = \overline{a_n \oplus b_{n+1}} \end{cases} \quad (20.)$$

from which we can deduce:

$$\begin{cases} a_n = \overline{U_n^0} \\ b_n = R_n^0 \oplus U_{n-1}^0 \end{cases} \quad (21.)$$

By substituting above values (21) in equations (19), we get the U_n^1 and R_{n+1}^1 values for the next iteration:

$$\begin{cases} U_n^1 = U_n^0 \oplus R_n^0 \oplus U_{n-1}^0 \oplus c_n \\ R_{n+1}^1 = (U_n^0 \oplus R_n^0 \oplus U_{n-1}^0) . (U_n^0 \oplus c_n) \end{cases} \quad (22.)$$

An implementation of such an accumulator stage for an iteration i is given fig. 6.

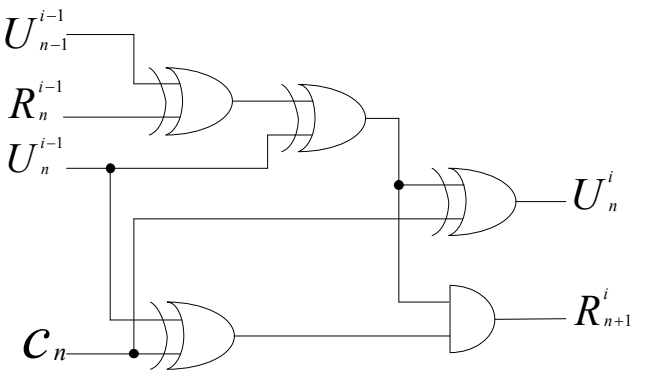

Figure 6. BSCB one bit accumulation stage

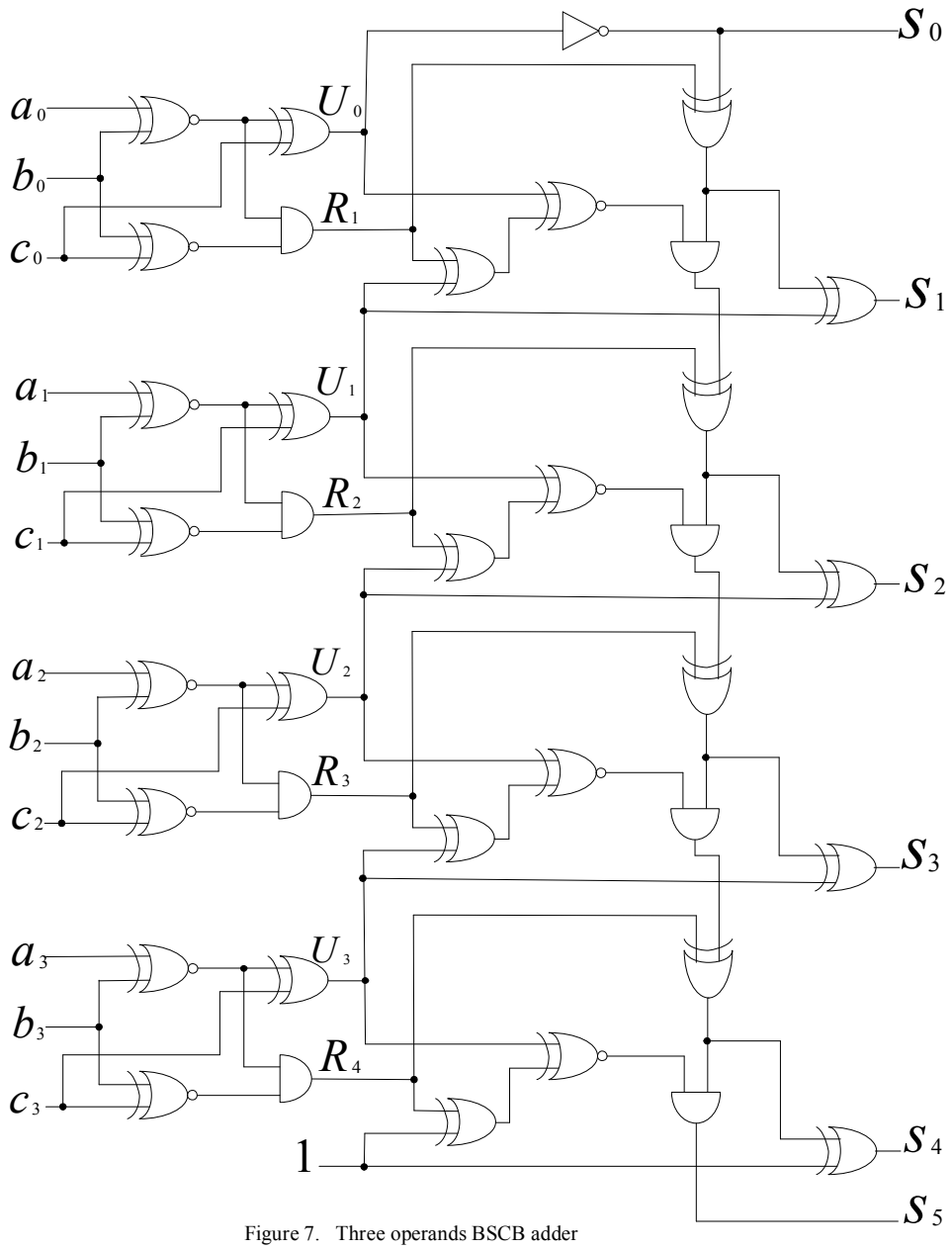

Figure 7. Three operands BSCB adder

IV. CMOS PROTOTYPE

A CMOS prototype carry-look-ahead adder was developed within the frame of a cooperative structure named Multi-Project-Chip (MPC) dedicated to student research projects.

This prototype was designed at ENSEEIHT [12].

The available technology was the 0,35µm CMOS from AMS. The goal was to compare two different carry-look-ahead designs in terms of area, speed and power consumption:

- A standard one composed of known AND, OR and EXOR gates,
- A second one implemented with AND gates and uniquely EXOR gates of the type described in [13] according to fig. 1.

Figure 8 and 9 show respectively the electrical and physical designs of the EXOR gate. It has a size of 23µm x 32 µm (736 μm^2). The four PMOS and four NMOS transistors are located respectively at the top and at the bottom of the design. All the layout distances are at the minimum size allowed by the design rule checker.

TABLE V. DESIGN COMPARISON

	Transistor count	*Size in µm*	*Area in mm^2*	*Prop. Time in ns*
Standard adder	106 PMOS 106 NMOS	361 x 67	0,024	6.31
Exor adder	114 PMOS 114 NMOS	361 x 82	0,029	6.58

TABLE VI. POWER CONSUMPTION (μW)

Freq. MHz	Standard adder		Exor adder	
	Binary count	*Gray count*	*Binary count*	*Gray count*
0.78	9.9	7.46	18.55	14.06
1.56	15.35	10.79	27.75	20.13
3.12	24.55	16.17	41.84	29.60
4.17	30.69	19.54	52.27	35.67
6.25	42.11	25.64	70.09	45.11
7.14	46.99	28.31	78.14	50.23
8.33	53.50	31.88	89.07	57.76
10	62.67	36.86	103.59	65.57
12.5	75.74	43.66	123.72	77.19

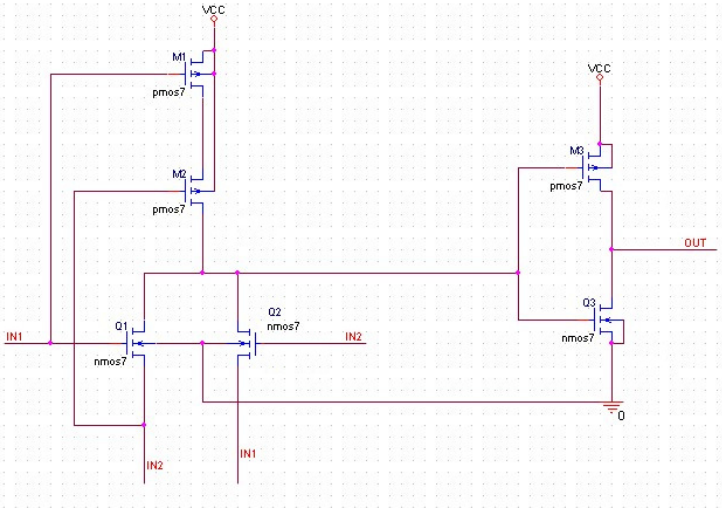

Figure 8. Exor gate

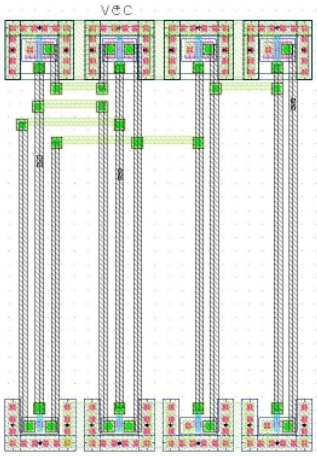

Figure 9. Exor gate design

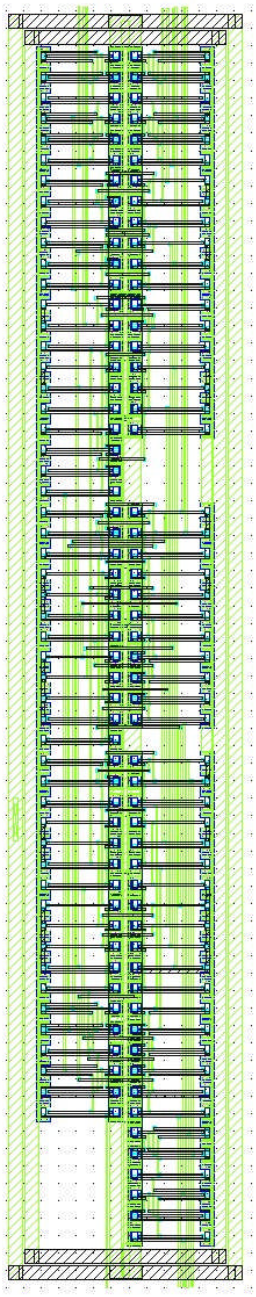

Figure 10. Standard 4 bits adder

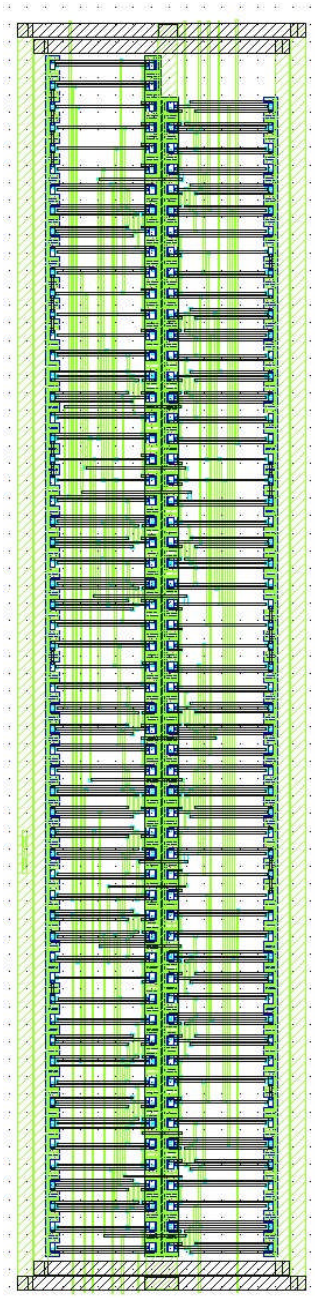

Figure 11. Exor 4 bits adder

The propagation times given in table V are averages of all the propagation times when the input counter is running over all the 512 combinations. The Exor adder is 4.52% slower, but his difference is not significant as both designs were not designed for speed and there are still improvements to be done for example in the final Exor tree.

Operating with a power supply of 3.28V the static power is 3.45nW for the standard adder and 3.54nW for the Exor adder. Table VI displays the power consumption of both adders at increasing frequency. It shows that the consumption is closely related to the frequency and in fact to the transistor commutations as the static consumption is almost the same for both designs. This is also enhanced by the results of the binary counting method compared to the gray counting method which generates less gate commutations.

V. CONCLUSION

New implementations of binary adders were proposed. The number system representation on which they rely is the binary stored-carry-or-borrow representation. It seems that this BSCB representation leads rather directly to Reed-Muller implementations of all known types of adders (carry-look-ahead adder, ripple-carry adder, half-adder, full adder).

A direct advantage of this kind of implementations is that the testability is improved over known implementations due to the use of EXOR gates as it was demonstrated by previous works. The related drawback is that the increase of gate commutations generates an increase of the power consumption New designs of Exor gates could be a way to explore.

Among related subjects that may be of interest, following items should be mentioned:

- Reliability which could be made easier to implement. For instance in the case of the carry-look-ahead adder the parities of the two inputs can be used to generate very easily the parities of the signal issued by the first EXOR stage of the adder,

- Reversible logic which could be also a promising way of investigations as more than 75% of the described implementations is made of EXOR gates.

ACKNOWLEDGMENTS

The author thanks Marie-Lise Flottes, Bruno Rouzeyre and Driss Aboulkassimi from LIRMM (Laboratoire d'Informatique, de Robotique et de Microéléctronique de Montpellier) for their valuable contributions to the testability study of the different adder implementations.

The author also acknowledges Hélène Tap-Béteille and Francis Bony from ENSEEIHT (Ecole Nationale Supérieure d'Electrotechnique, d'Electronique, d'Informatique, d'Hydraulique et des Télécommunications de Toulouse) for their works in respectively the design and the simulation/tests of the CMOS adder circuit.

REFERENCES

[1] Kai Hwang, *Computer Arithmetic: Principles, Architecture and Design,* John Wiley and Sons, pp. 69-96, 1979.

[2] Reto Zimmermann, *Binary Adders Architecture for Cell-based VLSI and their Synthesis,* Hartung-Gore, 1998.

[3] Behrooz Parhami, *Computer Arithmetic, Algorithms and hardware designs*, Oxford University Press, 2000

[4] Michael J. Flynn, Stuart F. Oberman. *Advanced Computer Arithmetic Design*, Wiley Interscience Publication, 2001.

[5] Israel Koren, *Computer Arithmetic Algorithms*, A.K. Peters Limited, 2002..

[6] Milos D. Ercegovac, Tomas Lang, *Digital Arithmetic,* Morgan Kaufmann, pp. 51-135, 2004.

[7] Jean-Michel Muller, *Elementary Functions, Algorithms and Implementation*, Birkhaüser, 2006.

[8] Seiji Kajihara, Tsutomu Sasao, "On the Adders with minimum tests" *Proceedings of the 5th Asian Test Symposium (ATS97),* 1997.

[9] Driss Aboulkassimi, Marie-Lise Flottes, Bruno Rouzeyre, "Etude de la testabilité d'additionneurs et multiplieurs" *LIRMM report*, (French) 2009, unpublished .

[10] Bernd Becker, "Efficient testing of Optimal Time Adders" *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1113-1121, 1988.

[11] Daniel Torno, "Additionneurs binaires ExorAnd", *SARL DANIEL TORNO internal report* (French), 2008, unpublished.

[12] Francis Bony, Hélène Tap-Béteille, "Réalisation et test d'additionneurs 4 bits en technologie CMOS submicronique" *ENSEEIHT report*, (French) 2008, unpublished.

[13] Hung Tien Bui, Abdul Karim Al-Sheraidah and Yuke Wang. "New 4-transistor XOR and XNOR designs" *The Second IEEE Asia Pacific Conference On ASICs*. Aug 28-30, 2000.