A CAD Framework for Generating Self-Checking[1] Multipliers Based on Residue Codes

I. ALZAHER NOUFAL, M. NICOLAIDIS
Reliable Integrated Systems Group, TIMA
France

Abstract

The basic drawbacks related to the design of self-checking circuits include high hardware cost and design effort. Recent developments on self-checking operators based on parity prediction compatible schemes allow us to achieve high fault coverage and low hardware cost in self-checking data paths for the majority of basic data path blocks such as, adders, ALUs, shifters, register files, etc. However, parity prediction self-checking multipliers involve a hardware overhead significantly higher than for other blocks. Thus, large multipliers will increase significantly the hardware overhead of the whole data path. Residue arithmetic codes allow to reduce this cost. The tools presented in this paper generate automatically self-checking multipliers using such codes. They complete our tools using parity prediction for various other blocks, and enable automatic generation of low cost self-checking data paths.

Keywords: *Self-Checking Circuits, Fault Secure Circuits, Multipliers, Residue Arithmetic Codes.*

I. Introduction

Self-checking designs achieve concurrent detection of errors produced by both permanent and transient faults. In the past this kind of circuits was dedicated to specific applications requiring high safety levels (e.g. Railway control [DUF96], control of critical functions in automotive [BOE97], etc.), or evolving in hostile environments (e.g. satellites). In addition, device shrinking, power supply reduction and increased operating speeds that accompany the process of very deep sub-micron scaling, affect adversely circuit noise margins [NIC 98a] [NIC 98b] [NIC 98c]. This makes circuits increasingly sensitive to various internal and external noise sources. We are approaching a point where soft error rates are becoming unacceptable. In this context, protection against soft errors will be necessary even for applications for which reliability is not a main concern. For instance, for CMOS processes below 0.1 μm, the. frequency of single event upsets (SEUs) induced by hot neutrons will become unacceptable even at ground level.

Due to the high clock frequencies this problem will no more affect memories only but also logic parts. The main problem will concern the logic parts, since memories can be protected efficiently by using error detecting and correcting codes. Self-checking circuits may provide a potential solution to this problem, since they can detect soft errors immediately as they occur. Thus, we can correct them by repeating the last operation. As a matter of fact, in the new context, the interest for self-checking design is enhanced drastically. Data paths are essential (and generally the biggest) logic parts of microprocessors and micro-controllers. As a matter of fact efficient self-checking data path design may have an important impact on the design of the next generations of ICs. Self-checking designs verifying the fault secure property guaranty that the output errors generated by the modelled permanent or transient faults are detectable. Thus, designs verifying the fault secure property will allow to achieve high levels of reliability.

The basic drawbacks related to the design of self-checking circuits include high hardware cost and high design effort. Using parity codes for checking memory systems and register files may guaranty the fault secure property and still maintain low hardware cost. However arithmetic operators produce output errors of random multiplicity which generally are not detectable by the parity code. Thus the parity prediction scheme [SEL68], [GAR68] does not achieve the fault secure property for these circuits. Recent developments on arithmetic operator design based on parity prediction [NIC 93] [NIC97a] [NIC97b] [DUA97] have demonstrated the feasibility of low cost fault secure, for many of the basic blocks used in data paths including: adders, ALUs, dividers and shifters. These solutions was integrated into a set of tools that generate automatically various versions for these blocks. These tools reduce drastically the design effort related to the implementation of a self-checking data path and make self-checking data paths attractive. However, in the case of multipliers, fault secure design based on parity prediction requires a hardware overhead in the range of 40% to 50% . This is much higher than the overhead required for the other types of blocks. In

[1] This Work was supported by MEDEA A401 project

the case of large multipliers it will affect adversely the hardware overhead required for the whole data path. As an alternative, residue arithmetic codes can be used to check the arithmetic operators [PET72] [AVI73] [SPA93] [SPA94]. These codes add in the information part a check part representing the modulo A of the information part. Usually A is of the form 2^k-1, and the resulting codes are known as low-cost residue arithmetic codes. These codes may ensure fault secureness in many arithmetic operator cases. In addition, the arithmetic code prediction for an adder or a multiplier requires compact hardware. It consists on a small arithmetic operator (adder, multiplier) which adds or multiplies operands of two or three bits length (the check parts of the operands), followed by a modulo A generator which computes the modulo A of the result. This hardware is constant (i.e. independent of the size of the arithmetic operator). Arithmetic code checking also requires an arithmetic code checker and some code translators. The translators (modulo A generators and parity generators) are required between the parity checked parts (memory system and some blocks of the data path) and the arithmetic code checked parts. The size of the checker and translators is proportional to the size of the operands and represent a high hardware cost. Because the extra parts are either of constant size or proportional to the size of the operands, while the size of the multiplier is proportional to the square of the size of the operands, for large multipliers the area overhead for the arithmetic coding can become very low. However for small and medium size operands the parity prediction will require a lower overhead. Thus, an efficient CAD system must includes tools for both schemes. To complete our software, this paper presents a set of tools allowing automatic generation of self checking multipliers based on residue arithmetic codes. The tools handle a wide range of multiplier structures.

II. Self-Checking Data Path

An example of a self-checking data path based on a BUS oriented architecture is shown in figure 1.This data path includes: a Carry Checking/Parity Prediction adder/ALU block based on the technique presented in [NIC93] [NIC97b], a parity prediction shifter implemented according to [DUA97], and a register file extended to include a parity bit to each register location. The functional blocks communicate through the data path BUSes A and B. The S-C shifter includes means for parity prediction, while the S-C adder/ALU includes means for carry checking and parity prediction. The register file and the BUSes include a parity bit. The data are checked during their transfer from one block to another by the two parity checkers connected to the BUSes. The selection of parity checking for the BUSes, and register file is obvious since we just need to add a parity bit, resulting to the lowest possible hardware cost. In addition this solution achieves fault secureness due to the bit slice structure of these parts, and also it is compatible with parity checked memory systems. The selection of parity prediction for shifters is also clear since this scheme achieves the fault secure property by means of low hardware cost [DUA97], while arithmetic code prediction for shifters requires very high hardware

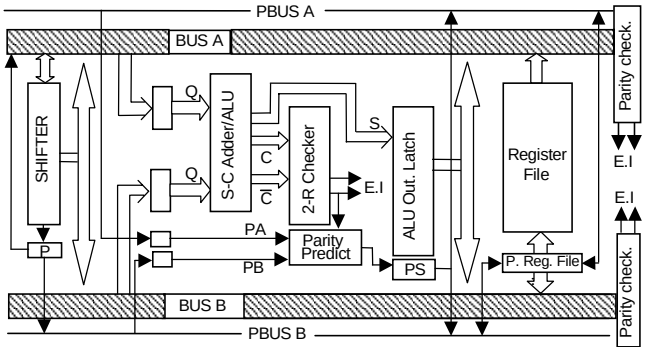

Figure 1: Self-checking data path

cost. The selection of the Carry Checking/Parity Prediction scheme for adders and ALUs is also clear since this scheme achieves the fault secure property for any adder/ALU design by means of low hardware cost [NIC93] [NIC97b]. On the other hand, arithmetic code prediction requires to use two arithmetic code generators to generate the check parts for the adder input operands, an arithmetic code checker to check the results, and a parity generator to make the results compatible with the other data path blocks. We also have to add the arithmetic code prediction circuit, which in the case of adders is simple but in the case of ALUs is very complex These extra blocks require a very high hardware cost, so that the use of arithmetic codes in the case of adders/ALUs has reduced interest.

The choice is less obvious in the case of multipliers. For these circuits the parity prediction solutions presented in [NIC97a] achieve fault secureness by means of a hardware cost of the order of 40% to 50%, which is significantly higher than in the case of adders/ALUs and shifters. On the other hand, although the use of arithmetic codes for multipliers requires the same input and output code translator and checkers as for adders, these blocks have a cost linear to the size of the operands, while the cost of the multiplier is linear to the square of the operands. Thus, for large multipliers the use of arithmetic codes may result on very low cost. As a matter of fact, an efficient set of tools must include generators of S-C multipliers based on both schemes. This way the user will be able to implement the less expensive solution depending on the size of the multiplier. Based on the solution described above, we have developed several parameterised macro-block generators. They generate all the basic blocks with parameterised size required for the design of the S-C data paths including: parity checkers, double rail checkers, parity prediction shifters, Carry Checking/Parity prediction adders and

ALUs for various ripple-carry and carry look ahead implementations, parity prediction for various multiplier structures and array dividers, as well as various multiplier structures based on arithmetic codes. The tool is developed in C and generates a netlist description of the operator in VHDL or Verilog HDL. This description can then be used by standard CAD tools to synthesize the layout of the self-checking design. The following sections present the tools allowing the generation of self-checking multipliers based on arithmetic codes.

III. Self-Checking Multipliers Based on Residue Arithmetic Codes

In the following, we discuss the fault secure implementation for the various structures of multipliers proposed in the literature and used in practice. The fault model includes faults affecting a single gate at a time. These faults will produce an error on the gate output. The error can be propagated through the subsequent gates until the multiplier outputs. To guaranty fault secureness, all output errors produced by this process must be detected by the residue arithmetic code. The fault model considered above can be extended to include distinct sets of gates. Thus, in the case of figure 2 we will consider any fault affecting the set of gates generating C and any fault affecting the set of gates generating S. In figure 3 we will consider any fault affecting the XOR gate generating P, the XOR gate generating S, or the set of gates generating C.

III.1. Array Multipliers

Array Multipliers are implemented by a partial product generator (AND gates) and a network of full and half adders. The network of adder cells is often implemented, as an iterative array of full and half adders, followed by a carry propagate adder (last row of the array), which is usually implemented as a ripple-carry adder (e.g. Braun multiplier, Pezaris multiplier, etc. [HWA79]).

For array multipliers the effects of an error in self-checking multipliers based on residue arithmetic codes are well known from the literature. To ensure the fault secure property for this scheme, the arithmetic value of the errors produced at the multiplier outputs must not be divided by the base of the residue arithmetic code. An error produced on one output of a full/half adder cell has the arithmetic value +, - 2^i, where i is the weight of the erroneous signal. This error is added to the result through the adder cells network, resulting on the same value of output error. This is a single arithmetic error and it is not divided by 3. Thus, if the cells produce single errors on their outputs (i.e. only the output *S* or only the output *C* is erroneous) we can use as check base A = 3, which corresponds to the cheapest check base (2 check bits). This situation will arise when there is no circuitry shared between the carry and sum outputs of the adder cell, as in the case of figure 2. On the other hand, when *S* and *C* share some gates, as in figure 3, both signals *S* and *C* can be erroneous at the same time. In this case the error can also take the values + 3. 2^i and - 3. 2^i. Then the check base A = 7 can be used to achieve fault secureness. However, in figure 3, the double error on

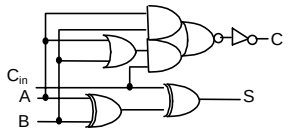

Figure 2: cell with no logic shared between The signals S and C (type 1)

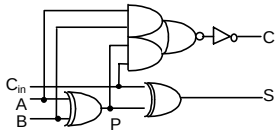

Figure 3: cell with some logic shared between The signals S and C (type 2)

signals *C* and *S* occurs when a fault on the shared logic produces an error on the signal *P*. This error is always propagated to *S* (since $S=P \oplus Cin$). Since $C = P.Cin+A.B$, this error will also be propagated to *C* only if $Cin = 1$ and $A.B = 0$. In this case we have $S = \overline{P}$ and $C = P$. Thus, the arithmetic values of this double error are $2^{i+1}-2^i = 2^i$ for $P = 0 \rightarrow 1$ and $-2^{i+1}+2^i = -2^i$ for $P = 1 \rightarrow 0$. In both cases we have a single arithmetic error. Therefor , using the check base A = 3 will achieve fault secureness for the cell of figure 3 too.

III.2. Fast Multipliers

III.2.1. Wallace Trees

For large multipliers the arrays of full and half adders will introduce large signal delay, i.e linear to the number of the multiplier inputs. For achieving logarithmic delay, the partial products generated by the AND gates are accumulated using Wallace trees [KOR93]. Although the structure of the network adder cells used in the Wallace trees is different than the one used in array multipliers, the errors produced to the multiplier outputs due to a fault in an adder cell are the same. This is because in both cases the adder network will add to the final result the arithmetic value of the error produced on the outputs of the faulty adder cell.

The outputs of the wallace trees feed a carry propagate adder. This adder can be implemented in a ripple-carry manner. This adder structure is a network of adder cells and the output error is again $\pm 2^i$. Thus the base 3 achieve fault secureness as discussed above. However, the ripple-carry implementation of the carry propagate adder is inefficient since the Wallace trees exhibit logarithmic

delay while the delay of the ripple-carry adder is linear. Thus, to improve the speed, the carry propagate adder must be implemented in a carry lookahead manner. In that case, the adder is not a network of full and half adder cells, and the choice of the base is less obvious. This problem is discussed below.

III.2.2 Base Selection for Fast Adders

The base selection for fast adders was discussed in details in [SPA93]. The problem here is that the
carries are generated by a carry lookahead function which has a structure radically different than the adder cell networks. Thus, the error propagation is also different, resulting on various error types. To determine the arithmetic values of the output errors, we can consider the following facts :
1. only faults on signals with divergent degree higher than one may give errors different than $\pm 2^i$.
2. consider an error on such a signal (e.g. signal w_i). This error can be propagated on carry c_i. Each carry c_j (j>i) functionally dependent on c_i is also functionally dependent on w_i (i.e. c_j can be expressed as a function of w_i and of other various signals). However, in the actual implementation we can have that all signals cj (j>i) are structurally dependent on w_i or only a subset of these signals are structurally dependent on w_i. In the first case an error on w_i is always propagated on an error of the form $\pm(2^{i+k}-2^{i+k-1}-2^{i+k-2}-\ldots-2^{i+1})= \pm 2^{i+1}$, which is a single arithmetic error. The errors in the second case can be obtained from the errors obtained from the first case $[\pm(2^{i+k}-2^{i+k-1}-2^{i+k-2}-\ldots-2^{i+1})]$ if one removes the errors on the carries which are not structurally dependent on w_i . The analytical description of these errors is given below :
The signals c_{i+k}, $k \in \{1, 2, \ldots, r\}$; c_{j1+k1} , $j_1>i+r$, $k_1 \in \{1, 2, \ldots, r1\}$; c_{j2+k2} , $j_2>j_1+r_1$, $k_2 \in \{1, 2, \ldots, r_2\}$; …; c_{jm+km} , $j_m>j_{m-1}+r_{m-1}$, $k_m \in \{1, 2, \ldots, r_m\}$, $j_m+r_m<n$ are structurally dependent on w_i and no other signal is structurally dependent on w_i . In this case the output errors are :
$\pm[a_0(2^{i+1}+2^{i+2}+\ldots\pm 2^{i+k})+a_1(2^{i1+1}+2^{i1+2}+\ldots\pm 2^{i1+k1})+$
$a_2(2^{i2+1}+2^{i2+2}+\ldots\pm 2^{i2+k2})+\ldots+ a_m(2^{im+1}+2^{im+2}+\ldots\pm 2^{im+km})]$,
where $a_0, a_1, \ldots, a_m \in \{0,1\}$.
In addition to this relationship we have the following conditions :

1. $a_{q+1}=0$ if $a_q=0$.
2. $a_{q+1}=0$ if the sign of 2^{iq+kq} is negative.
3. the sign of 2^{iq+kq} is always negative if $k_q<r_q$.

In order to exploit these results our software explores the carry lookahead network to establish the set of carries that are dependent on each signal w_i . From these sets, the analytical description of the output errors is generated. Then, the tool searches for the smallest odd integer that does not divide any of these errors values. This number can be used as check base to achieve fault secureness for the multiplier that includes the particular adder structure on its output stage. Because the low-cost arithmetic codes (check base of the type 2^k-1) usually require simpler hardware than other check bases, then, if the smallest check base identified by the above procedure is not of the type 2^k-1, the tool also searches for the smallest check base of this type. This way we can generate the hardware for both bases and select the most compact one.
The macro-block generators for fast adder modules in our set of tools generate the most efficient adder structures proposed in the literature, including : Kogge&Stone [KOG73], Han&andCarlson [HAN87], Brent&Kung [BRE82], Sklansy [SKL60], and an adder using Carry Look Ahead Units [HWA79]. Table 1 shows the bases found for the different structures used. For Kogge-Stone and Han-Carlson cases we give the minimal base and the minimal low cost base that ensure fault secureness.

Check Base	16 bits	32 bits	64 bits	128 bits
Kogge&Stone	2^6-1	37	37	37
		2^{12}-1	2^{12}-1	2^{12}-1
Han&Carlson	2^4-1	2^6+1	2^6+1	2^6+1
		2^8-1	2^8-1	2^8-1
Brent&Kung	7	7	7	7
Sklansky	3	3	3	3
Carry Lookahed Units	3	3	3	3

Table 1 : Check bases found for different structures of fast carry determination.

III.2.3. Booth Multipliers

In Booth multipliers, the number of partial products is reduced by one half by means of encoding the biggest input operand. So, the area cost for the overall circuit is also reduced. For these multipliers, beside the problem of check base selection for the fast adder (which might be used at the last stage), local errors in the Booth encoding may produce output errors which need a base > 2^{n-1} to be detected (where n is the size of the biggest input operand). In [SPA96], the duplication of the encoder was proposed, then errors in encoding logic are detected by using a double-rail checker. It was also shown [SPA96] that some local errors in the network of full and half adders produce overflow errors at the output of the multiplier (since two's complement numbers are treated). In this case the lowest check base achieving fault secureness is 7. For the base 3, undetectable errors may occur only for one input combination, and for a very small number of faults. Thus the use of the check base 3 will slightly reduce reliability. As a matter of fact, in our experiments we have used both 3 and 7 values for the check base.

IV. Modulo Generators

The residue generator is an essential building block for arithmetic error detecting codes. Because the output errors are of the type $\pm 2^i$, only odd check bases are of interest. For low-cost arithmetic codes (check base of type 2^k-1), modulo residue generators can be efficiently implemented by exploiting particular properties of modulo arithmetic applied to these codes. In fact for $A=2^k$-1, 2^{km} modulo A=1 for any integer m. Thus, the residue of a binary number can be obtained by splitting it into bytes of k bits and performing modulo A addition of these bytes. Modulo A addition of two k-bit bytes is performed by using a k-bit adder in which the carry output is injected on the carry input (carry end-around adders). This is because 2^k modulo (2^k-1)=1. Thus, a tree of k-bit carry end-around adders can be used to build the residue generator for the check base $A=2^k$-1. These results are known from the very early developments on residue arithmetic codes (e.g. see [AVI73]). However in several multiplier cases, residue generators for other check bases can be of interest.

In [PIE94] new procedures for synthesizing residue generators and Multi-Operands Modular Adders (MOMA's) were presented. First, the periodic properties of the series of powers of 2 taken mod A, previously observed and exploited for $A=2^a$-1, were extended to any odd A. Then, due to periodicity, it was shown that a Carry Save Adder/Carry Propagate Adder (CSA/CPA) network with end-around carry can be built for any odd base A.. These new generators are shown faster and less costly than conventional designs. The simpler designs correspond to the bases of the type 2^k-1. Such an example for k=3 is shown in figure 4.a. The period of the powers of 2 modulo 7 is 3. Thus, this example exploits the fact that the residue modulo 7 of 1, 8, 64,**…** is equal to 1. Thus X_0, X_3 and X_6 are added by using a full adder. Similarly, the residue modulo 7 of 2, 16, 128, **…** is equal to 2. Thus a full adder is used to add X_1, X_4, and X_7, and so on. Another simple case corresponds to the bases of type 2^k+1. Such an example for k=3 is shown in figure 4.b. The half-period of powers of 2 modulo 9 is 3. Thus, this example exploits the fact that the residues modulo 9 of 1, 8, 64, **…** are respectively +1, -1, +1, **….** Thus , X_0, $-X_3$ and X_6 have to be added. Similarly the residues modulo 9 of 2, 16, 128 are respectively 2, -2, 2, **….** Thus X_1, X_4, X_7 have to be added. To avoid subtraction which may require complex circuits, the above operations are performed by using a full adder and inverting the bit that has to subtracted. In this operation, each inverted bit add an error of $+1x2^i$ (i.e. +1 for X_3, +2 for X_4 and +4 for X_5). This error is corrected by subtracting a constant modulo A at the end. In the example we have to subtract 8 modulo 9 or equivalently to add 1 modulo 9. For bases different than $A=2^a$-1 and $A=2^a$+1 the design is more complex. It employs some ROM's and a Modular Adder. For more details see [PIE94]. Sharting from this analysis we have implemented systematic procedures that allow to implement residue generators for any check base.

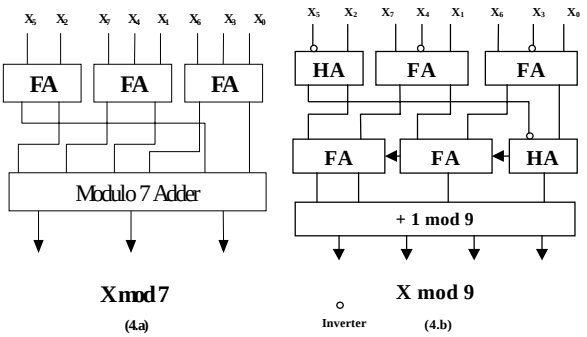

Figure 4 : Schematic block for 8-bits number modulo 7 and modulo 9.

V. Self-Checking Multiplier System

The block diagram of the self-checking multipliers based on residue arithmetic codes is shown in figure 5. It is composed of the multiplier block, two modulo A generators to generate the check parts of the input operands (these check parts are not provided by the data path which is checked by the parity code), a Modular Multiplier to generate the check part of the result of the multiplication, an arithmetic code checker to check the result of the multiplication against its check part (such a checker is not provided by the data path), and finally a parity generator to generate the parity of the result for use by the other data path blocks. The Modular Multiplier is composed of AND gates and a Modular Adder [PIE94] in the case of low cost bases and of a multiplier and a modulo generator in other cases. The arithmetic code checker is composed of a modulo A generator to generate the residue of the multiplier output, a translator [NIK88] to overcome the problem of the two representations of zero (not necessary when using not low-cost bases), and a double-rail checker. The design is pipelined as shown in the figure 5, by inserting a set of latches on the outputs of the multiplier and on the outputs of the Modular Multiplier. Thus, the results of the multiplier are checked during the subsequent clock cycle. So, no performance penalty will be introduced if none of the blocks placed beyond or below the latches has a delay longer than the multiplier itself. This case however happens in small or medium size multipliers(8x8, 16x16) using check bases different than 2^k-1 and 2^k+1. In such situations the modulo A generator placed on the outputs of the check-parts multiplier is transferred to the front of the arithmetic code checker (i.e. below the latches). This balances better the delays of the different parts and reduces the performance penalty. This solves problem for most cases but for check bases different than 2^k-1 and 2^k+1 we still introduce performance penalty as illustrated from the results shown in the next sections. To generate the self-checking multiplier system, we have

implemented a set of tools for determining the check bases, and various parameterised

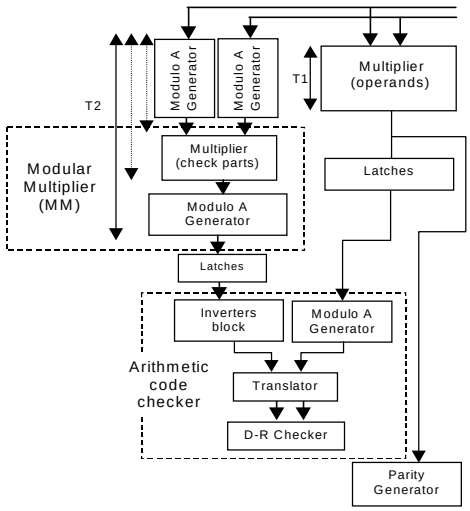

Figure 5: Block diagram of the S-C multiplier system based on the residue arithmetic codes.

macro-block generators that generate the different blocks shown in figure 5 (modulo A generators for any check base and data length, translator, 2-rail checker and multipliers). In particular our tools generate various multiplier structures such as array multipliers, Wallace trees multipliers and Booth multipliers, using various carry propagate adders such as ripple-carry, Kogge&Stone, Brent&Kung, Sklanski, and carrylook ahead units. The tool is implemented in C and generates netlist description in VHDL and Verilog HDL.

VI. Experimental Results

The tool was used to perform experiments for various multiplier sizes and structures. Area and time were calculated using the Synopsys software. The results are shown in tables 2 to 7. In tables 2, 4 and 6 T1 represents the worst case delay for the multiplier and T2 the worst case delay for the extra blocks (see figure 5). For Braun multiplier, the Full Adder choice impacts drastically the speed of the multiplier and so we have reported results for the two types of full adder (see figures 2 and 3). In the other cases, the two full adders produces almost the same delay and so we reported results for type 2 full adder which is smaller. The area overhead for 8x8 multipliers is quite high. In this case the parity prediction scheme [NIC97a] will be preferred. For bigger multipliers the area overhead is reduced significantly going down to 5% or 6% for 64x64 multipliers. This illustrates the superiority of the residue checked scheme in the case of big multipliers. Tables 4 and 6 concern the case of multipliers using Wallace trees and fast carry propagate adders on their last stage. The type of the adder structure determines the check base required to achieve fault secureness. In tables 4 and 6 we can observe that for small operands size (8x8 and16x16), the use of Kogge-Stone and Han-Carlson

Performance Penalty	8x8		16x16		32x32		64x64	
	T1 (ns)	T2/T1 %	T1 (ns)	T2/T1 %	T1 (ns)	T2/T1 %	T1 (ns)	T2/T1 %
FA type1	29,9	0,0	64,7	0,0	135	0,0	282,9	0,0
FA type2	34,4	0,0	74,1	0,0	155.3	0,0	321,6	0,0

Table 2 : Performance penalty for Braun multiplier.

Area Overhead(%)	8x8	16x16	32x32	64x64
FA type1	47.3	21.5	10.3	5.0
FA type2	54.4	25.0	12.0	5.9

Table 3 : Area overhead results for Braun multiplier.

Performance Penalty		8x8		16x16		32x32		64x64	
		T1 (ns)	T2/T1 %	T1 (ns)	T2/T1 %	T1 (ns)	T2/T1 %	T1 (ns)	T2/T1 %
Kogge & Stone	37			34,7	75,2	54,1	23,8	91,0	0,0
	2^{12}-1	23,5	58,7 (2^6-1)	34,7	67,4	54,1	7,4	91,0	0,0
Han & Carlson	2^6+1			39,3	22,4	58,7	0,0	95,3	0,0
	2^8-1	26,2	8,0 (2^4-1)	39,3	22,4	58,7	0,0	95,3	0,0
Brent & Kung	7	27,3	0,0	40,3	0,0	62,4	0,0	102,3	0,0
Sklanski	3	26,1	3,8	37,9	0,0	58,3	0,0	96,6	0,0
CLA units.	3	26,3	3,0	39,3	0,0	59,4	0,0	98,8	0,0

Table 4 : Performance penalty for Wallace multiplier.

Area Overhead (%)		8x8	16x16	32x32	64x64
Kogge & Stone	37		100.3	35.8	11.9
	2^{12}-1	82.4 (2^6-1)	68.8	23.3	9.0
Han & Carlson	2^6+1		42.6	17.5	7.6
	2^8-1	55.0 (2^4-1)	40.7	16.5	7.4
Brent & Kung	7	53.4	26.0	12.8	6.3
Sklanski	3	43.9	22.0	11.1	5.6
Carry look Ahead.	3	45.6	22.7	11.4	5.7

Table 5 : Area overhead results for wallace multiplier.

adders might compromise the speed of the multiplier due to the selection of check bases that result on modulo A generators with long delay. In some cases we changed the pipeline position of the check-part in order to not compromise the speed. An example of that is the case of Wallace 32x32 with Kogge-Stone adder with check base=37, where the latches will be positioned on the outputs of the modulo generators of the two operands. Note that this change also implies an extra cost of 6 latch cells. For check bases not of the type 2^k-1 or 2^k+1, the

hardware overhead and performance penalty can be quite high. Thus, in that case, we do not stop with the lowest check base but we continue our search until finding the smallest base of the type 2^k-1 or 2^k+1. If this base of the form 2^n-1 the solution corresponds to the duplication checking. Among the 5 fast adder cases, only the Kogge&Stone adders give a minimal check base (37) not of the type 2^k-1 or 2^k+1. For this adder we have also found that the base 2^{12}-1 guarantees fault secureness. This base, although much bigger than 37, gives better results as shown in tables 5 and 7. Also for the three adders, the lowest check base is a low-cost one (i.e. of the type 2^k-1). Finally for the Han&Carlson adder, the lowest check base is 2^6+1. In this case also we have searched for a low-cost check base and found the 2^8-1 one. This base gives slightly better results. From the results we observe the following :

1. The selection of the adder structure impacts the selection of the check base. This last may have a considerable impact on hardware cost since modulo A generators are very expensive for some check bases. Also they may be very slow making some blocks slower than the multiplier itself, thus impacting adversely the performance of the whole design. The Kogge&Stone adder which gives significant improvement on the multiplier speed when compared with the adder using carry look ahead units, is giving slower self-checking 8x8 and 16x16 multipliers. This is due to the check bases required to achieve fault. In addition, the carry lookahead units adders are more compact than any other adder. Thus they will be preferred in most cases to any other adder scheme. The Kogge&Stone adder is becoming of interest for 32x32 and 64x64 multipliers, in applications where speed improvements are important. For instance, for the 32x32 multiplier using the Kogge&Stone adder, the self-checking solution requires an area of 6.42 mm^2 and has an operation speed of 55.5 ns. The adder using carry lookahead units requires an area of 5.508 mm^2 and offers a speed of 59.4 ns. Thus by paying an additional 16.6% of area, the Kogge&Stone multiplier offers a 7% speed improvement
2. Note that the area overhead for implementing the self-checking solution is becoming very low as the multiplier size is increased. For instance, for the 16x16 Wallace multiplier using the carry look ahead units, the area overhead is 22.7% and becomes as low as 5.7% for the 64x64 multiplier. At the same time no performance penalty is introduced. This low-cost makes self-checking design quite attractive even for applications not requiring very high reliability levels.

The results for Booth-Wallace multipliers are shown in tables 6 and 7. The area cost is lower than the area cost required for both Braun multipliers and Wallace trees multipliers. This is true for both the simple and self-checking multipliers (except the 8x8 case). In addition the speed performance is better than in the case of Wallace trees multipliers. Finally the area overhead of the self-checking version is increased due to the duplication checking of the Booth encoder part. However it still remains acceptable for the 16x16 case (e.g. 33.5% for carry lookahead units case) and decreases down to 9.5% for the 64x64 multiplier.

Performance Penalty		8x8		16x16		32x32		64x64	
		T1 (ns)	T2/T1 %	T1 (ns)	T2/T1 %	T1 (ns)	T2/T1 %	T1 (ns)	T2/T1 %
Kogge & Stone	37			34.0	78,8	46.2	45,0	67.3	4,0
	2^{12}-1	25.9	44.0 (2^6-1)	34.0	70,9	46.2	25,8	67.3	0.0
Han & Carlson	2^6+1			37.5	27,2	49.4	9,7	70.4	0.0
	2^8-1	26.8	5,6 (2^4-1)	37.5	28,3	49.4	14,2	70.4	0.0
Brent & Kung	7	27.7	0.0	39.8	0.0	54.3	0.0	78.4	0.0
Sklanski	3	26.0	4,2	36.0	0.0	49.0	0.0	71.6	0.0
	7	26.0	0.0	36.0	0.0	49.0	0.0	71.6	0.0
Carry Look Ahead units	3	25.9	4,6	39.7	0.0	50.8	0.0	75.2	0.0
	7	25.9	0.0	39.7	0.0	50.8	0.0	75.2	0.0

Table 6 : Performance penalty for Booth-Wallace multiplier.

Area overhead (%)		8x8	16x16	32x32	64x64
Kogge & Stone	37		126,8	50,7	18,3
	2^{12}-1	92,4 (2^6-1)	88,8	34,1	14,3
Han & Carlson	2^6+1		56,9	26,5	12,1
	2^8-1	25,2 (2^4-1)	54,7	25,2	11,7
Brent & Kung	7	63.0	37,4	20,3	10,2
Sklanski	3	54,5	32,5	18,2	9,3
	7	62,5	36,5	19,9	10,1
Carry Look Ahead units	3	55,6	33,5	18,8	9,5
	7	63,7	37,6	20,7	10,3

Table 7 : Area overhead results for Booth-Wallace multipliers

In the results presented, the delay of the parity generator is not considered. This circuit adds several levels of XOR gates increasing the total delay. Although the faster XOR tree is the logarithmic one, this is not the network giving the best results, due to the unbalanced delays on the outputs of the multiplier. In the case of the multipliers having a linear delay (array multipliers, multipliers using a ripple-carry adder at their last stage, etc.), the best solution is to use a linear XOR tree. In this case and for any multiplier size, only one two-input XOR gate is added in

the critical path of the block composed by the multiplier and the parity tree. In fast multipliers (using Wallace trees and fast adders), again the logarithmic XOR tree does not provide the best solution. By adapting the tree structure to the delays of the multiplier array, the extra delay can be reduced by one or two XOR levels. The obtained results show approximately a delay of 18% for the 16x16 fast multipliers using Wallace trees, 14% for the 32x32 fast multipliers and 8.8% for 64x64 fast multipliers. However, when speed is a main concern this performance penalty can be avoided by sending the results of the multiplier to their destination as soon as they are ready, and sending their parity several nanoseconds later. This is possible because no block in the data path needs to receive the parity bit at the beginning of its clock cycle. This solution will require to add some latches within the parity generator (e.g. 5 latches for the 16x16 multiplier), and implement the parity slice of the data path by using a delayed clock. The details of this implementation are not given here for space reasons.

VII. Conclusions

In this paper we presented a tool for automatic generation of self-checking multipliers based on residue codes. It includes : a tool for computation of the check base required for achieving fault secureness, a tool for building residue generators for any check base and various macro-block generators producing a wide variety of multiplier structures. Experimental results show that the fault secure property can be achieved with low hardware cost, especially for large multipliers. This software completes our tools for automatic generation of self-checking data paths. They include generators for self-checking adders, ALUs, shifters, register files, dividers and multipliers, all based on parity coding. Thus, self-checking data paths are generated by means of low hardware cost and design effort. This kind of tools is becoming increasingly important, in a context where the number of applications requiring high levels of reliability is rapidly increasing and where the shrinking of device size in IC technologies will make mandatory the protection against transient errors.

References

[AND71] D. A. ANDERSON, "Design of self checking digital networks using coding techniques". Urbana, CSL/University of IIIionis, Sept. 1971 (report n. 527).

[AVI73] A. Avizienis, "Arithmetic Algorithms for Error-Coded Operands" IEEE Trans. on Comput., Vol. C-22, No. 6, pp.567-572, June 1973.

[BRE82] R. P. Brent, H. T. Kung, A Regular Layout for Parallel Adders", IEEE Transactions On Computers, Vol. C31, pp 261-264, March 1982.

[BOE97] E. Boehl, Th. Lindenkreuz, R. Stephan, "The Fail-Stop Controller AE11", ITC pp.567-577, Nov. 1997.

[DUA97] R.O. Duarte, M. Nicolaidis, H. Bederr, Y. Zorian, "Efficient Fault-Secure Shifter Design", European Design & Test Conference, Paris, March 1997.

[DUF96] Jen-Louis Dufour, "Safety Computations in Integrated circuits", VTS. pp. 169-172, April 28-May 1, 1996.

[GAR68] O.N. Garcia, T.R.N. Rao T, "On the method of checking logical operations", Proc. 2nd Annual Princeton Conf. Inform. Sci. Sys., pp. 89-95 (1968).

[HAN87] T. Han, D. A. Carlson, "Fast Area-Efficient VLSI Adders", Proc of 8th Symposium on Computer Arithmetic, pp. 49-56, May 1987.

[HWA79] K. Hwang, Computer Arithmetic, Principles, Architecture and Design, John Wiley and Sons, New York, 1979.

[KOG73] P. M. Kogge and H. S. Stone, A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, IEEE Transactions on Computers, Vol. C-22, No.8, pp. 786-792, August 1973.

[KOR93] I. Koren, Computer Arithmetic Algorithms, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[NIC93] M. Nicolaidis, "Efficient Implementation of Self-Checking Adders and ALUs", Proc. 23th Fault Tolerant Computing Symposium, Toulouse France, June 1993.

[NIC97a] M. Nicolaidis, R.O. Duarte, S. Manich, and J. Figueras, "Achieving Fault Secureness in Parity Prediction Arithmetic Operators", To appear in IEEE Design &Test of Computers.

[NIC97b] Nicolaidis M., "Carry Checking Parity Prediction Adders and ALUs", submited to IEEE Trans. on VLSI Systems.

[NIC 98a] M. Nicolaidis, "Scaling Deeper to Submicron: On-Line Testing to the Rescue", In proceedings FTCS-28, pp. 299-301, June 1998, Munich.

[NIC 98b] M. Nicolaidis, "Design for Soft-Error Robustness to Rescue Deep Submicron Scaling", In proceedings ITC 98, October 1998, Washington DC.

[NIC 98c] M. Nicolaidis, " On-Line Testing for VLSI: State of the Art and Trends", Integration: the VLSI Journal, Elsevier, Special Issue on "VLSI Testing toward 21 Century", Autumn 1998.

[NIK88] D. Nikolos, A. M. Paschalis, and G. Philokyprou, "Efficient Design of Totally Self-Checking Checkers for all Low-Cost Arithmetic Codes", IEEE Trans. On Comput, vol. 37, No. 7, pp. 807-814, July 1988.

[PET72] W. W. Peterson, E. J. Weldon, Error-Correcting Codes, second Ed., The MIT press, Cambridge, Massachusetts, 1972.

[PIE94] S. J. Piestrak, "Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders", IEEE Trans. On Comput, vol. 423, No. 1, pp. 68-77, Jan. 1994.

[SEL68] F. F. Sellers, M. Y. Hsiao and L. W. Bearnson, Error Detecting Logic for Digital Computers, New-York : Mc GRAW-HILL 1968.

[SKl60] J. Sklansky, "Conditional-Sum Addition Logic", in IRE Transactions on Electronic Computers, Vol. EC-9, No. 2, pp 226-231, June 1960.

[SPA93] U. Sparmann, "On the Check Base Selection Problem for Fast Adders", Proc. 11th VLSI Test Symp. Atlantic City, NJ, April 1993.

[SPA94] U. Sparmann, S. M. Reddy. "On the Effectiveness of Residue Code Checking for Parallel Two's Complement Multipliers" Proc. 24th Fault Tolerant Computing Symposium, Austin Texas, June 1994.