Fault-Secure Shifter Design: Results and Implementations

Ricardo O. Duarte[1], M. Nicolaidis[2], H. Bederr[3], Y. Zorian[4]
[1,2] TIMA- 46, Av. Félix Viallet - 38031 Grenoble - France
[3] Texas Instruments - 06271 - Villeneuve Loubet Cedex - France
[4] Lucent Technologies - Princeton, N.J. 08540 - USA

Abstract

Self-checking designs will gain increasing interest in industrial applications if they satisfy the following requirements: high fault coverage, reduced hardware cost and reduced design effort. This work is aimed to reach these requirements for the design of self-checking shifters and is part of a broader project concerning the design of self-checking data paths.

1. Introduction

Self-checking design [2] is an on-line testability technique, that implements functional blocks delivering outputs belonging on an error detecting code. A checker monitoring this code performs error detection concurrent to the normal circuit operation. Due to this concurrent error detecting ability self-checking (S-C) circuits is of high interest for applications requiring high levels of reliability. S-C circuits will gain increasing industrial interest if they can be implemented with reduced hardware cost and design effort and they can offer high fault coverage. This work concerns the design of low cost, high fault coverage self-checking shifters. In order to achieve the reduced design effort the proposed solutions are implemented into a S-C shifters macro-block generator. This work is part of a broader effort concerning the development of low cost, high fault coverage solutions for S-C data paths, and the implementation of these solutions into a CAD tool. [3],[5],[6]

2. Shifter implementation

Shifter units can be used for many different purposes: a simple division or multiplication by 2, an inversion of all bits or a simple signed arithmetic operation. Shifters can be implemented in different formats; such as barrel-shifters or multiplexer based. This work considers implementations using standard cells because they are becoming predominant in industrial context and also can be easily automated through the use of a HDL.

Shifters can be implemented in their simplest form allowing to perform a one-position shift-left or shift-right. This case results on low hardware cost but requires k clock cycles to perform a k-positions shift. Alternatively shifters allowing to perform a shift of any number of positions per clock cycle increase performance significantly but of course hardware cost is also increased. According to the system requirements either of these solutions can be selected, so that the tool must include macro-blocks generators for both cases. Also the tool is able to provide shifters performing any combination of the following operations: rotation, left or right logic shift and arithmetic shift.

Since we have decided to work in a standard cell environment our basic option concerns shifters based on multiplexers. Area/performance efficient shifters designs [4] are considered below. An illustration of this implementation is shown in figure 1.

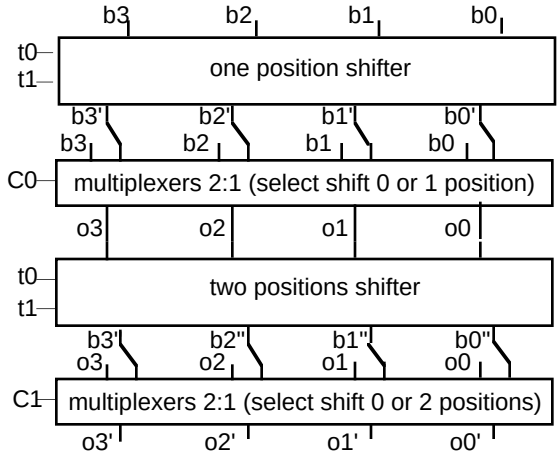

Figure 1. Shifter design implementation.

In this design the arithmetic value of the signals $C_{log(n)-1}$, ..., C_1, C_0 determines the number of positions that the input data have to be shifted. Thus the number of shifted positions is equal to $\sum_{i=0}^{log(n)-1} C_i 2^i$. This equation can be implemented by cascading a number of log(n) shifters, each one shifting a fixed number of positions. That is, the i^{th} shifter performs a shift of 2^i

[1] Under grant supported by CAPES-COFECUB, BRAZIL

ED&TC '97 on CD-ROM
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for fee or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. © 1997 ACM/0-89791-849-5/97/0003/$3.50

positions. A row of 1-out-of-2 multiplexers controlled by the signal C_i, selects the outputs of the i^{th} shifter for C_i = 1 or the inputs of the i^{th} shifter for C_i = 0. This implementation is illustrated for n = 4 in figure 1. For a shifter that executes only one type of operations the fixed position shifters (FPS) consist only on a routing circuit. That is the i^{th} FPS routes the r bit position to the (r-i) modulo n bit position. For a shifter that executes various operations each FPS is controlled by the control signals that determine the type of the operation to be performed. The present work is illustrated by considering the four most usual operations (i.e. rotation, left logic shift, right logic shift and arithmetic shift). The operation to be performed is determined by the values of two control signals t1,t0 : t1t0 = 00 for right rotation (ROT), t1t0 = 01 logic left shift (LLS), t1t0 = 10 for right logic shift (RLS) and t1t0 = 11 for right arithmetic shift (RAS). In this case each FPS is implemented using n 1-out-of-4 multiplexers that select the signal corresponding to the operation determined by the values of t1t0. For illustration, figures 2 and 3 present the FPS for one-position and two-positions shifts concerning a shifter of 4 bits.

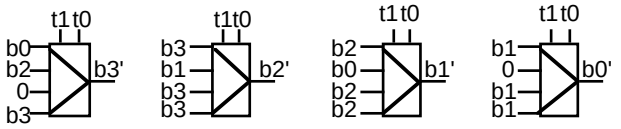

Figure 2. A four-bit shifter of one-bit position shift.

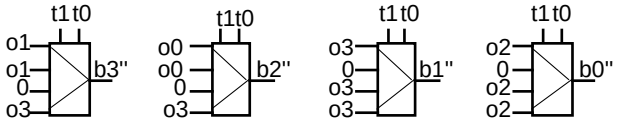

Figure 3. A four-bit shifter of two-bit position shift.

3. Self-checking shifter

The self-checking implementations for shifters considered here are based on parity prediction. This allows to be compatible with the self-checking data paths generators that we are developing. These data paths are based on parity checking. This solution allows to achieve low hardware cost since parity code use a single check bit.

In order to achieve high fault coverage the shifters will be designed to be fault secure. That is under any single fault in the shifter the errors generated on the shifter outputs are single and thus they are detected by the parity code.

3.1. Parity prediction circuit

For rotate operations the values of the bit positions are cyclically shifted so that any bit position value of the initial operand can be found in another position. This solution results on parity preservation. For predicting the parity of the other operations in a shifter of n bits, let us consider p_j to be the parity of the j less significant (rightmost) bits of the input operand, and p_j' to be the parity of the j most significant (leftmost) bits of the input operand. During a shift-left operation of k bits positions, the n-k right-most positions of the input operand are shifted into the n-k left-most positions of the result. The k left-most positions of the input operand are lost. The k right-most positions of the result are loaded by k 1's or k 0's (say k d's) according to the kind of shift operations. From that the parity of the result Pr can be predicted in two different ways:

$$\text{k even: } Pr = Pi \oplus P_k' \text{ (Eq.1)},$$
$$\text{or } Pr = P_{(n-k)} \text{ (Eq.2)}$$

$$\text{k odd: } Pr = Pi \oplus P_k' \oplus d \text{ (Eq.3)},$$
$$\text{or } Pr = P_{(n-k)} \oplus d \text{ (Eq.4)}$$

which Pi is the parity of the input operand.

Similarly for a shift-right or an arithmetic shift operation of k positions we can find:

$$\text{k even: } Pr = Pi \oplus P_k \text{ (Eq. 5)},$$
$$\text{or } Pr = P_{(n-k)}' \text{ (Eq. 6)}$$

$$\text{k odd: } Pr = Pi \oplus P_k \oplus d \text{ (Eq. 7)},$$
$$\text{or } Pr = P_{(n-k)}' \oplus d \text{ (Eq. 8)}$$

In these equations d is 0 for left and right logic shifts and it is equal to the most significant bit b_{n-1} for right arithmetic shift.

When we design a shifter that is able to shift any number of positions (that is k can take any value between 1 and n-1), the relationships (1) to (8) show that we need to compute either the values p_k or the values p_k' for $k \in \{1,2, ..., n-1\}$.

The cheapest way for generating the values p1 to pn, consists on implementing the parities in a linear manner as show in figure 4. Using this implementation, no extra XOR gates have to be added in the bus checker. Unfortunately this solution introduce a delay linear to the length of the shifter. This will considerably reduce the speed of the parity checker and of the shifter.

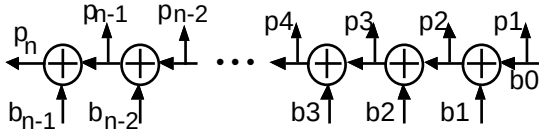

Figure 4. Parities generation: linear approach.

To cope with this problem the XOR trees generating the signal p_j have to be implemented as binary trees. These trees can be merged to reduce the gate count. They are constructed around the tree generating pn (to be said

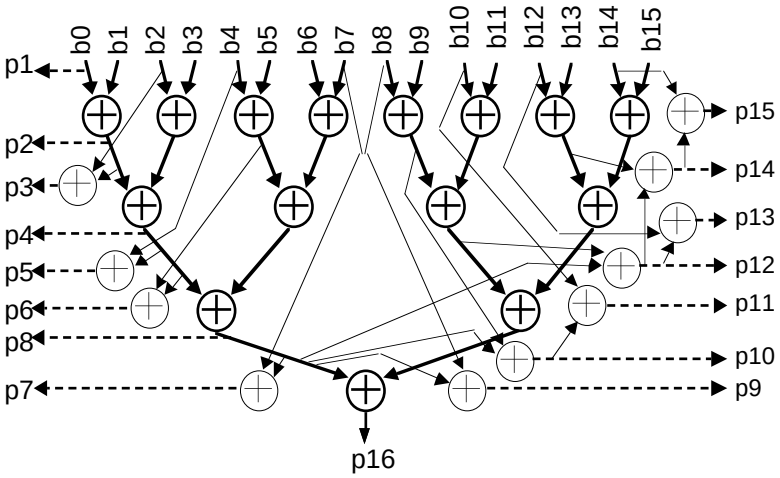

Figure 5. First approach for the parities generation using the XOR binary tree.

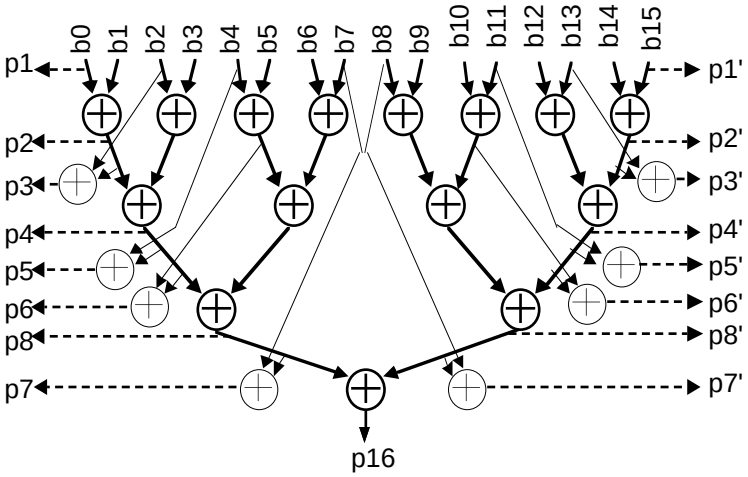

Figure 6. Optimised logarithmic parity tree

basic tree). The basic tree will be in fact the parity checker of the bus that provides the input operand to the shifter [3],[5]. Figure 5 shows an example for n = 16, where the basic tree correspond to the bold XOR tree arrangement. The basic tree is in fact the parity checker of the bus that provides inputs to the shifter. This solution is generalised trivially for any n being a power of 2 (which is the usual case for data paths).

When n is not a power of 2, we can still construct a circuit based on a number of inputs n' > n (n' being a power of 2) and then remove the extra inputs and the related gates.

We can remark in figure 5 that the delay of signals p11, p13, p14 and p15 is higher than the logarithmic one. It is desirable to reduce this delay into the logarithmic one (minimum delay) and also reduce the gate count of the solution of figure 5. This is possible by manipulating the equations (1) to (8) we can reduce both the delay and gate count of the solution presented in figure 3. For simplicity let us consider the case $n = 2^r$. In the case of left logic shift we can use the equations:

$$\text{for } k < 2^{r-1},\ Pr = Pi \oplus p_k' \quad \text{(Eq. 9)}$$

$$\text{for } k \geq 2^{r-1},\ Pr = p_{(n-k)} \quad \text{(Eq. 10)}$$

In the case of right logic shift and arithmetic shift we can use the equations:

$$\text{for } k < 2^{r-1},\ Pr = Pi \oplus p_k \oplus d \quad \text{(Eq.11)}$$

$$\text{for } k \geq 2^{r-1},\ Pr = p_{(n-k)}' \oplus d \text{ for k odd.} \quad \text{(Eq. 12)}$$

where d is 0 for right logic shift and b_{n-1} for arithmetic shift.

These equations lead to the solution of figure 6 which is faster and smaller than the one of figure 5.

The block diagram for the proposed solution to be included into a datapath is presented in figure 7. The input operands of the shifter are provided by the bus A. The decoding circuit receives its inputs from the parity tree (basic tree and extra XOR gates) of bus A, as well as the parity Pa of bus A. From these signals it predicts the parity Pr of the shifter outputs, that will be latched at the

same time as the shifter's latches. Since the checker of bus A is used for both predicting the parity Pr and checking the outputs of the shifter when they transit in bus A, a fault in the parity tree of this checker could result on error masking (i.e. the fault can affect the predicted parity in a first time and then mask this error when it checks the result of the shifter).

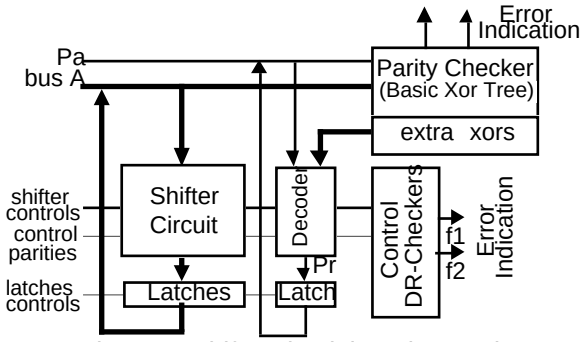

Figure 7. Shifter circuit in a data path

This does not happen however, since when an error is generated to some of the signals p_j due to a fault in an XOR gate of the basic tree, then, this error necessary propagates through the XOR gates of the basic tree until the signal pn (where n is the number of bits of the shifter - p16 in our example). Thus, an error indication is generated immediately on the signals Pa, pn before any erroneous parity is predicted for the shifters.

Control checkers are also needed in the datapath, to avoid faults caused by the control signals of the shifter.

3.2. Decoding circuit

The decoding circuit (figure 8) implements the parity prediction equation corresponding to the actual operation performed by the shifter. Each of the equations corresponding to the operations that can be performed by the shifter is implemented by a particular circuit. The output of the circuit corresponding to the actual operation of the shifter is selected by a multiplexer controlled by the signals ti, ..., t0. In the case considered here we have: t1t0 = 00 for right rotation (ROT), t1t0 = 01 left logic shift (LLS), t1t0 = 10 for right logic shift (RLS) and t1t0 = 11 for right arithmetic shift (RAS).

During rotation the parity is preserved. Thus the parity Pi (Pa, represented in figure 7) of the input operand is transferred to Pr. During left logic shift the equations (9), (10) are used. Thus we have:

$$k \le n/2 - 1 \text{ (i.e. } C_{\log(n)-1} = 0)$$
$$Pr = p_k' \oplus Pi \quad \text{(Eq. 13)}$$

$$k \ge n/2 \text{ (i.e. } C_{\log(n)-1} = 1)$$
$$Pr = p_{(n-k)} \quad \text{(Eq. 14)}$$

These equations are implemented by the block A in figure 8.

For left logic shift operation the equations (11), (12) are used with d=0. We obtain:

$$k \le n/2 - 1 \text{ (i.e. } C_{\log(n)-1} = 0)$$
$$Pr = p_k \oplus Pi \quad \text{(Eq. 15)}$$

$$k \ge n/2 \text{ (i.e. } C_{\log(n)-1} = 1)$$
$$Pr = p_{(n-k)}' \quad \text{(Eq. 16)}$$

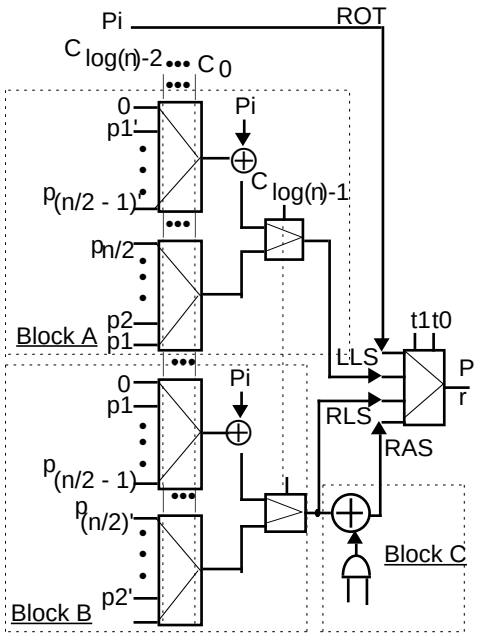

Figure 8. Decoding circuit

These equations are implemented by block B in figure 8. Finally for arithmetic shift the equations are the same as for right logic shift, excepting that d can be equal to 1. Since the k left most bits are replaced by b_{n-1}, d will be equal to (k x b_{n-1}) mod 2. Thus, for k odd (i.e. $C_0 = 1$) we have $d = b_{n-1}$, for k even ($C_0 = 0$) we have d = 0. This corresponds to $d = C_0 \wedge b_{n-1}$ and is implemented by block C.

4. Fault secure property

In this section we show that the proposed design is fault secure [1], [7], that is, for any single fault either the output of the shifter is correct or the produced error is detected by the parity prediction.

Faults on the parity prediction circuit:

Any fault on the parity prediction circuit can affect only the predicted parity Pr, the outputs of the shifter is correct. Thus this error is always detected by the parity

checker of the bus (figure 7).

Faults on the shifter:

1 - Fault affecting an 1-out-of-2 multiplexer (figure 1). In this case the output of one multiplexer can be erroneous. This error is shifted or not shifted by the next stage FPS and MUXes, creating a single error on the outputs of these MUXes. The process is repeated until the primary outputs of the shifter, resulting on a single error detectable by the parity.

2 - Fault affecting a 1-out-of-4 MUX (figure 2 and 3). The output of one of these MUXes can be erroneous. This error can be propagated on the output of the corresponding 1-out-of-2 MUX and it is detectable as in case 1.

3 - Fault affecting an input of an FPS. This input enters two 1-out-of-4 MUXes (see figure 2 and 3). However the corresponding inputs of these MUXes are selected by different values of t1 t0. Thus the error is propagated to the output of only one of these multiplexers at a time. As in case 2 this error is always detectable.

4 - The analysis (in point 3) holds for all situations excepting for faults affecting the most significant bit (MSB) during an arithmetic shift. In case of a k-positions arithmetic shift the MSB of the input data is propagated to the k+1 LSB positions of the output data. Thus the following faults can create multiple output errors during arithmetic shifts:

a - A fault on the input data MSB (b_{n-1}).

b - A fault on an 1-out-of-2 MUX generating the MSB of the next stage FPS, or a fault on the MUX of an FPS generating the MSB output of this FPS, or a fault on the input of an FPS.

Case a - fault:

During a k positions arithmetic shift, the MSB (b_{n-1}) is shifted to the k+1 bit position of the output data (as in the case of other shift right operations), but it is also charged to the k LSB positions of the output data. Thus, an error on the input MSB is propagated to k+1 output errors.

If k is odd, we have seen (section 3.2) that the predicted parity is given by $Pr = p_k \oplus Pi \oplus b_{n-1}$, or by $Pr = p_{(n-k)}' \oplus b_{n-1}$, and the error on the MSB affects it. Thus the total number of errors on the outputs and on their parity Pr is odd (k+2) and it is detectable.

If k is even we have seen (section 3.2) that the predicted parity is given by $Pr = p_k \oplus Pi$, or by $Pr = p_{(n-k)}'$. Thus the error on b_{n-1} does not affect it. Therefore the total number of errors is again odd (k+1) and it is detectable.

In fact what happens is that in a k-positions arithmetic shift, the input data MSB appears once on the output data as for a normal right shift. In addition it appears in the k LSB positions. Because of that, if k is odd the MSB is also used to modify the parity Pr . If k is even there is no need to use the MSB in the parity computation. Thus in both cases the input data MSB affects an odd number of bit positions in the output data and their parity Pr.

Case b- faults:

The fault does never creates an error on Pr since the circuit generating Pr is completely independent from the shifter circuit.

All the faults of case b- create a single error on the MSB input of some FPS (excepting the input of the 0-level FPS which is never affected by these faults).

We can check that, excepting the 0-level FPS which performs a one position shift, all other FPS perform a shift of an even number of positions. Thus whatever are the values of the control lines Ci, the total number r of positions that the input of any other FPS is shifted until the outputs of the shifter, is always even. The above mentioned error on the MSB input of an FPS will be propagated on r+1 outputs. Since r is always even, r+1 is odd and the error is always detectable.

Interconnection faults:

While this analysis shows that fault secureness holds also for arithmetic shifts, there could be situations where fault secureness will be lost if interconnections are not carefully implemented. If an interconnection branch of the input data MSB feeds an even number of inputs of the 0-level FPS and of the parity prediction circuit, then, an open on this branch will affect only an even number of these signals. We can check that in this case the total number of output errors is even. Similarly if an interconnection branch of the MSB input of an FPS feeds an even number of inputs of this FPS, an open of this branch will create an even number of output errors. Branches of this kind should be avoided in the design. For instance, the input data MSB should divided in a single point into three branches, one going to the parity prediction circuit, one to the first input of the first FPS and one to the second input of the first FPS (as shown in figure 9a. Similarly the branches of the MSB input of any other FPS are implemented as in figure 9b.

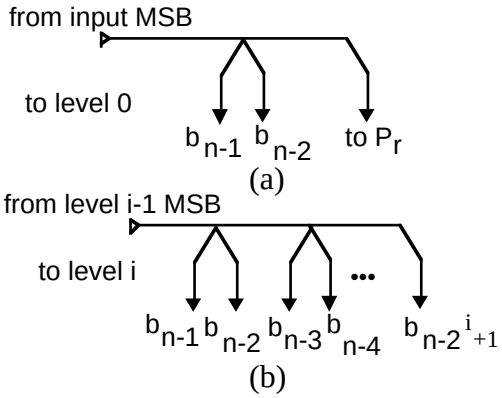

Figure 9. Implementation of the interconnections.

5 - Fault affecting the control signals t1, t0, C_{logn-1}, ..., C_1,C_0. These signals control all the cells of each stage. Thus a fault on these signals can produce multiple output errors that can be left undetectable by the parity code.

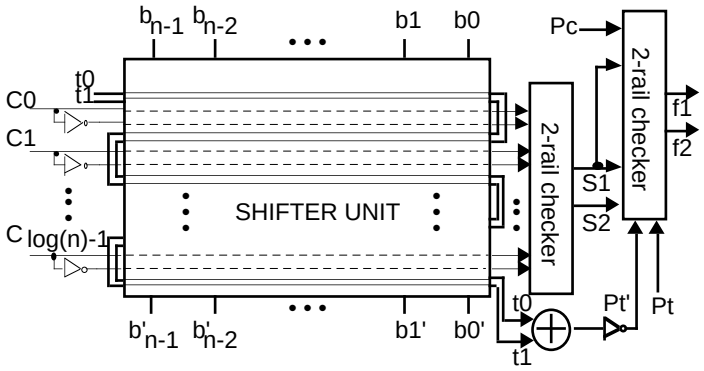

Figure 10. Control signals checker.

Since the signals C_i are divided into two branches C_i and $\overline{C}_i$ both these branches have to be checked, requiring a double-rail checker. On the other hand the original signals C_i must also be checked to verify if they bring the correct values. The simplest way is to check the parity Pc of these signals, provided that Pc is generated by the blocks generating the signals C_i. This requires a parity checker. To reduce the cost of this solution we will use a double-rail checker (see figure 10) to check the branches C_i and $\overline{C}_i$. The outputs S1, S2 of this checker indicate an error on these branches. At the same time the output S1 computes the odd parity of the signals C_i [5]. Thus the signals Pc, S1 indicate errors on the original signals C_i.

On the other hand an XOR gate computes the odd parity Pt' of signals t1, t0. Thus the signals Pt (generated by the circuit generating t0 and t1) and Pt' indicate an error on t1, t0. The three error indications are compressed by a double-rail checker into a single error indication (signals f1, f2).

5. Implementations and experiments

Based on the solution described above, we have developed a parameterised macro-block generator. It can generates n-bit shifters that are able to perform one-position shift or 0 to n-1 positions shifts, according to the specifications of the user. Also the user can specify the shifter to perform any combination of the four shift operations described above.

The tool is developed in C and generates a netlist description of the shifter in VHDL or Verilog HDL. This description can then be used by standard CAD tools to synthesise the layout of the self-checking shifter. We have used the tool to generate various shifter sizes. Numerical results were obtained, from shifter netlists mapped into the standard cell library of ES2 - ecpd10 - 1.0 µm CMOS technology.

The hardware overhead for shifters performing any number of shift positions in one clock cycle and the four basic operations: rotation, left and right logic shifts and arithmetic shift, is shown in table 1. This overhead is about 20 % for a 8-bit shifter and decreases to 15 % for a 32-bit shifter. This cost is quite good given the advantages of concurrent checking. In particular it will not increase the hardware overhead of the self-checking data paths generated by our tools, which is lower than 20 % for the other macro-block generators such as adders, ALUs, register files, buses and their checkers.

Note that if the checker implements only rotation, right logic shift and arithmetic shift, the hardware overhead is reduced by about 35 %. This is due to the elimination of block A in figure 8. This block for medium and large shifters represents the half area of the decoding circuit.

We considered in the results of area overhead presented in table 1: the decoding circuit, the extra XORs gates and the latches. The area overhead of the checkers (control and bus checkers) are not considered, because they do not belong exclusively to the shifter, but to the whole datapath. They are normally computed when all elements of the datapath are included (register files, buses, ALUs).

Table 1. Shifters of 0 to n-1 bits positions shift: area overhead.

Nb. of bits	%
4	29,2
8	21,7
16	18,0
32	15,6

Table 2. Shifters of 0 to n-1 bits positions shift: worst case delay

Nb. of bits	absolute value (ns)	%
4	2,15	22,0
8	1,33	8,9
16	1,02	2,8
32	0,88	1,0

Table 2 presents the worst case delays introduced by the parity prediction circuitry. Although the delay is significant for a 4-bit shifter (22 %), it becomes very low or insignificant for larger shifters (i.e. 9 % for the 8-bit shifter down to 1 % for 32-bit shifter).

6. Conclusions

This paper presents a technique for implementing fault secure shifters based on parity prediction. This design achieves high levels of reliability by means of moderate hardware cost. A macro-block generator has been implemented and integrated in our tools for automatic generation of self-checking data paths. The tool was used to implement self-checking shifters of various sizes. These implementations show that the area overhead is about 20% or less for medium and large shifters, presenting a good overall performance for fault secure circuits.

References

1. D.A. Anderson, "Design of self-checking digital networks using coding techniques", *Coordinates. Science Laboratory, Report R/527. University of Illinois*, Urbana, September 1971.
2. W.C. Carter, P.R. Schneider, "Design of dynamically checkers computers", *in Proc. 4th Congress IFIP, vol.2*, Edinburgh, Scotland, Aug. 5-10 1968, pp. 878-883.
3. B. Hamdi, H. Bederr, M. Nicolaidis, "A tool for automatic generation of self-checking data paths", *13th IEEE VLSI Test Symposium*, Princeton N.J., April-May 1995.
4. K. Hwang, *Computer Arithmetic, Principles, Architecture and Design*, John Wiley and Sons, New York - 1979.
5. M. Nicolaidis, "Efficient implementation of self-checking adders and ALUs", *Proc. 23th Fault Tolerant Computing Symposium*, Toulouse France, June 1993.
6. M. Nicolaidis, S. Manich, J. Figueras, "Achieving fault secureness in parity prediction array arithmetic operators", *1996 - European Design and Test Conference ED&TC*, Paris, March 1995.
7. J.E. Smith, G. Metze, "Strongly fault secure logic networks", *IEEE Transactions on Computers*, June 1978.