

**FAULT-TOLERANT WSI ARRAY PROCESSORS:
THE USE OF BERGER CODES IN PARALLEL ARITHMETIC-LOGIC UNITS**

VINCENZO PIURI

Department of Electronics, Politecnico di Milano
piazza L. da Vinci 32, I20133 Milano, Italy

ABSTRACT

Design of fault-tolerant WSI array processors is discussed: the use of Berger codes is presented to introduce a unifying approach both for arithmetic and logic operations. Detection of unidirectional errors and costs of the coding technique are evaluated in architectures based upon parallel computing units. Different structures are proposed for concurrent error detection and fault-localisation.

INTRODUCTION

The high computation needs of many applications often require special-purpose architectures for parallel computation. These systems became increasingly attractive since they may be implemented by means of VLSI and WSI technologies at reasonable costs and with competitive performance. Array processors are an example of these architectures: they consist of a number of identical processing elements connected by a regular interconnection grid [1]. The characteristics and the flexibility of arrays are related to the structure of the processing elements and the interconnection network.

Reliability and continuous operation of these complex architectures are one of the most important problems, in particular for mission-critical applications or when maintenance is too difficult, expensive or impossible. Present VLSI and WSI integration technologies cannot completely prevent the occurrence of life-time faults due to production defects or to workload stresses. High reliability and availability may be achieved by means of techniques for defect and fault tolerance, which usually involve detection of errors due to a fault, correction of errors, localisation of the faulty processing elements and reconfiguration of the system to bypass the faulty modules. Fault tolerance and system performance depend on the effectiveness and efficiency of such activities.

In this paper we propose and evaluate a data coding technique based upon Berger codes [2] to introduce concurrent error detection and fault localisation in WSI array processors. Our approach is particularly suited for VLSI, ULSI and WSI implementations: it provides the distributed hardware support for concurrent identification of life-time faults, while the survival of the overall system and the exclusion of faulty elements can be achieved at array level through reconfiguration of interconnections [3] by using the information produced by the distributed diagnosis. Verification of the state of the system without suspending the nominal computation may be useful to reduce testing and repair time and, thus, to increase the availability of the system.

In section 3 we show the low redundancy requirements of Berger codes and the high error-detection capabilities with respect to other classes of codes for arithmetic-logic operations. Different architectures are presented and evaluated in section 4: error detection is introduced at different levels to protect the single processing element or the whole array. Fault localisation is considered in section 5 to generate information for implementing efficient reconfiguration strategies.

BERGER CODES FOR ARITHMETIC-LOGIC OPERATIONS

The characteristics of the operations performed in the processing elements must be accurately considered in choosing the fault-tolerance technique. Efficient data coding must in fact be specialised for the arithmetic-logic operations executed in the processing elements. Different arithmetic codes were proved effective to protect arithmetic operations [4] [5] [6], since they exploit the properties of arithmetic operations to obtain the output redundancy bits directly from the input data through

arithmetic operations. These codes do not guarantee the correctness of the other kinds of operations (e.g. logical operations) since the relationships between inputs and outputs are intrinsically different from the arithmetic functions. On the other hand, some classes of parity codes (e.g. single parity code [7], Read-Muller code [8], checksum code [9]) were studied for logic operations. Unfortunately, all previous techniques are not able to assure high protection both of logical and arithmetic operations.

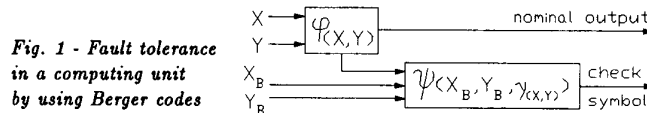
This paper discusses the use of Berger codes as a unifying approach to concurrent error detection in array processors. We show in particular that these codes may be used to protect both arithmetic and logic operations, minimising the redundant circuits and the computational delay. A single coding technique for correctness verification of all operations may be exploited to design small arithmetic-logic units since encoders and checkers are required only at array borders to generate the coded inputs and to verify the nominal results. These advantages may be effectively used to implement WSI array processors with concurrent fault tolerance.

Berger codes [2] are a class of parity-based codes: a codeword is obtained from the nominal data by joining the sequence of the nominal bits with the sequence of the redundancy bits (check symbol). The latter bits may be defined by means of two different encoding schemes:

- 1) the check bits are the binary representation of the number of 0's in the information bits, or
- 2) the check bits are equal to the complement of the number of 1's in the nominal data.

The codes generated by these definitions and their error detection capabilities are equivalent [2], therefore, we consider only the first encoding scheme. For a nominal data $X = \{x_1, x_2, \dots, x_n\}$, the codeword is $[X | X_B]$, where the Berger check symbol $X_B = \{x_{n+1}, \dots, x_{n+k}\}$ is given by $X_B = \sum_{i=1}^k x_{n+i} = \sum_{j=1}^n \bar{x}_j = n - \sum_{j=1}^n x_j$, where $k = \lceil \log_2(n+1) \rceil$. These codes are optimal since they have a high information ratio $IR = n/(n + \lceil \log_2(n+1) \rceil)$. If $n = 2^k - 1$, the code is a maximal-length Berger code since it fully exploits the information redundancy and the related hardware redundancy.

Berger codes are systematic (i.e. redundancy bits are separated from the information bits), but they are not separate codes (i.e. redundancy bits cannot be processed independently from the information bits). In fig. 1 the general scheme of a fault-tolerant computing unit based upon Berger codes is shown. The function $\phi(X, Y)$ is executed on the uncoded data X and Y to produce the result R . Partially in parallel, a function $\psi(X_B, Y_B, \gamma(X, Y))$ generates the check symbol R_B of the nominal result R . The systematic property of these codes allows to avoid decoders since nominal results may be easily obtained by separating the information bits from the redundancy bits of the coded results. The error occurrence may be identified by comparing the actual check symbol R_B with the expected check symbol R_E : the latter can be independently computed from the nominal result R by using the formal definition of the adopted Berger code.



The generation of the check symbol of the result for all usual arithmetic-logic operations may be achieved by using simple arithmetic-logic functions of the operands [10] [11]. The main results are presented in table 1 for two's complement arithmetic units: since we consider arrays in which results of any operation are used in subsequent operations, we assume that no overflow occurs without loss of generality. Otherwise, the most-significant carry must be recirculated to adjust the least-significant redundancy bits. Note that the check symbol of the quotient of the division cannot be predicted since the quotient is in the form of carry bits: error detection can be performed by verifying the check symbol of the residue.

Berger codes are able to verify the presence of all unidirectional errors [2]. The class of errors detected by these codes is considerably large and it is greater than those detected by many parity and arithmetic codes. The capability of identifying multiple errors becomes very interesting to implement WSI array processors by using highly-dense integration technologies. In this case, the probability of multiple faults should not be neglected.

Tab. 1 - Berger check symbol for arithmetic-logic functions

class	function	check symbol	class	function	check symbol
arith.	X+Y	$X_B + Y_B - n + N(C)^\dagger$	logic	0	n
	X-Y	$X_B - Y_B - n + N(C) + c_{n+1} - 1^\ddagger$		1	0
	X*Y	$nX_B + nY_B - X_B * Y_B +$ $-n^2 + \lceil \log_2(2n+1) \rceil + N(C)^\dagger$		X	X_B
	X/Y	$X_B - (n+3)Y_B - nQ_B + 2Y_BQ_B +$ $+2Y_Bq_1 - (n-2)q_1 - 2 + N(C) + N(R^*)^\ddagger$		$\bar{X}$	$n - X_B$
rotate	$\leftarrow$	X_B	logic	$X \wedge Y$	$X_B + Y_B - (X \vee Y)_B$
	$\rightarrow$	X_B		$\bar{X} \wedge Y$	$-X_B + Y_B - (\bar{X} \vee Y)_B$
logical	$\leftarrow$	$X_B - c_{in} + x_n$		$X \wedge \bar{Y}$	$X_B - Y_B - (X \vee \bar{Y})_B$
	$\rightarrow$	$X_B - c_{in} + x_1$		$\bar{X} \wedge \bar{Y}$	$-X_B - Y_B - (X \wedge Y)_B + n$
arith.	$\leftarrow$	$X_B + x_n$	logic	$X \vee Y$	$X_B + Y_B - (X \wedge Y)_B$
	$\rightarrow$	$X_B + x_1$		$\bar{X} \vee Y$	$-X_B + Y_B - (\bar{X} \wedge Y)_B + n$
shift	$\leftarrow$	$X_B + x_n$		$X \vee \bar{Y}$	$X_B - Y_B - (X \wedge \bar{Y})_B + n$
	$\rightarrow$	$X_B + x_1$		$\bar{X} \vee \bar{Y}$	$-X_B - Y_B + (X \vee Y)_B + n$

$^\dagger N(C)$ is the number of 1's in the set C of internal carries of the units.

$^\ddagger N(R^*)$ is the number of 1's in the set R^* of the $(n+1)$ most-significant bits of the residue outputs in a non-restoring divider.

The effectiveness of error detection in arithmetic-logic units via Berger codes may be extended to protect cascades of units and, thus, the processing elements and the whole structure of the array processor. In fact, it is possible to shown that the presence of an error is not masked by subsequent operations performed on a faulty intermediate result. This characteristic allows to greatly reduce the number of checking circuits since only the outputs of the processing element or those of the whole array need to be verified. Checking of intermediate results inside each processing element could be applied to decrease the error latency, but it is neither required to assure the correctness of the array outputs nor to preserve the detection capabilities of the code.

HARDWARE ARCHITECTURES: ARRAYS AND UNITS

Array processors are composed by a high number of relatively simple processing elements connected by an interconnection network [1]. The massive computational throughput is due to the capability of exploiting the parallelism of computation on data. Our attention is focused upon dedicated and semi-programmable arrays, i.e. upon architectures in which the processing elements execute a single hard-wired sequence of operations or a sequence fetched by an host computer. In both of these architectures inputs of the array may be supplied by an host computer or by custom interfaces (e.g. the A/D converter in a radar system). In particular, in this paper we consider arithmetic-logic arrays (i.e. arrays capable of executing arithmetic-logic functions) composed by parallel computing units. For simplicity sake, only linear and rectangular topologies are discussed. An example of dedicated array is presented in fig. 2 for band-matrix multiplication.

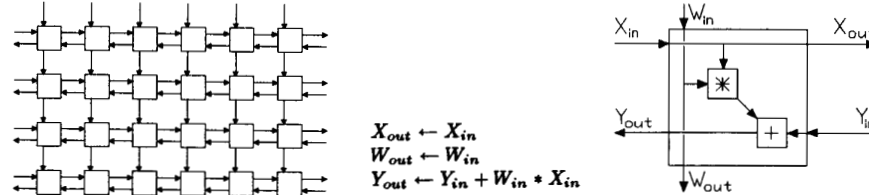


Fig. 2 - A rectangular array processor

The encoder for the Berger codes must compute the number of 0's in the input data. Since we consider parallel units, the encoder can be implemented by using a parallel counter with complemented inputs. Due to the high number of bits that must be treated, an area-efficient implementation of such counter is given by a regular tree of full and half adders [11]. The use of a custom circuit for a counter with more than three inputs should be avoided because of the silicon area. Decoders are not required for Berger codes since they are systematic codes: decoding consists therefore in considering the information bits and in neglecting the check bits.

Checkers may be obtained by using the cascade of an encoder and a binary comparator. The encoder produces the expected check symbol of the result under test, while the comparator verifies the congruence between the expected check symbol and the actual check symbol generated by the redundant arithmetic-logic units. In literature totally self-checking checkers are available, e.g. in [12] partitioning of Berger codewords is used to reduce the area and the checking time.

The general structure of a computing unit (i.e. an arithmetic or logic unit) for operands represented in a Berger code has been presented in fig. 1. The nominal computation is performed by a traditional arithmetic-logic unit, while the check symbol is computed from the check symbols of the operands and the internal carries of the nominal unit by applying the rules of tab. 1. Examples of the implementation of such redundant units are shown in fig. 3.

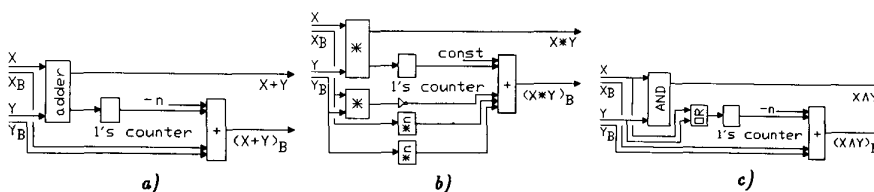


Fig. 3 - Redundant arithmetic and logic units: adder (a), multiplier (b), logic and (c).

In these units the different components of the formulas for generation of the actual check symbol of the output must be added together. An efficient architectural solution consists in the use of a multioperand carry-save adder; fig. 4 shows the complete implementation of this circuit for addition with 32-bit uncoded operands. A great reduction in the computation time and in silicon area may be achieved since not all output bits of such circuit must be computed. For example, the check symbol of the result of an addition is represented in the same number of bit of the input check bits: adders marked by "*" can be omitted, while adders marked by "+" can be simplified.

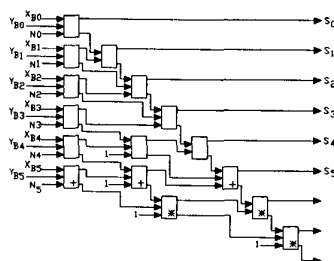


Fig. 4 - Multioperand carry-save adder

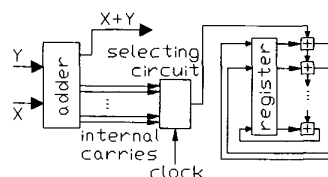


Fig. 5 - The pipelined 1's counter

Further area reduction may be achieved by considering the structure and the computation of the specific application and architecture. For semi-programmable arrays some groups of homogeneous arithmetic and logic units may be merged to reduce the overall silicon area. For example, addition and subtraction may be implemented in a single unit by introducing few additional logic gates [10]. In dedicated arrays different operations may be fused in a single functional block to reduce

the area and the computation delay. For example, the cascade of the multiplier and the adder for matrix multiplication (fig. 2) may be substituted by a single inner-product unit, in which the partial products of the multiplication and the second term of the sum are added by a network of full adders. In such circuit space is saved because the full adders are better exploited and the parallel counters, which compute the number of internal carries in addition and in multiplication, may be unified in a single counter. A small further reduction of the silicon area can be achieved by "squeezing" the parallel counter and the multioperand carry-save adder in a single circuit [10]. Unfortunately, there is no general structured design methodology since this compaction is strictly related to the specific application and, in particular, to the number of bits of the operands.

Pipelined computation of the number of 1's may reduce the silicon area, but increases the computational delay. An example is shown in fig. 5 for the 1's counter that must be used to perform addition with coded operands. Instead of waiting that all internal carries of the adder are computed, it is possible to test the value of each carry as soon as it is produced by the basic arithmetic unit: the possible 1's are thus immediately counted without waiting for the completion of nominal addition. To implement this technique it is necessary to design a clocked scheme which extracts the carries one at a time from the basic arithmetic unit when they are steady. For n -bit operands, a k -bit adder is required to count the input 1's, while a k -bit register is used to store the partial result. The selecting circuit provides one carry at a time for the counter: it could be implemented by means of a multiplexer, but it should require a large silicon area to route the selecting signals. A smaller solution consists in adopting a shift register initialised to 1 to mask the internal carries of the basic arithmetic unit; a clock signal imposes the proper timing to the shift register, whose content is moved one position on the left side at each clock cycle. This additional clock signal is tailored to guarantee the correctness of data storing in the counter register. To reduce the computational delay, it is possible to overlap testing of a new internal carry of the basic arithmetic unit and propagation of the carry inside the counter. To such purpose the carries in the counter must be latched. At each cycle the least-significant half adder of the counter receives a new internal carry and produces the new least-significant bit of the number of 1's and the carry that must be propagated inside the counter itself. Such carry is latched to provide the correct input for the adjacent half adder during the subsequent cycle and so on iteratively. Another improvement in the computational delay may be achieved in computing units which produce groups of internal carries at the same time (e.g. multipliers): in this case it is possible to count the number of 1's in each group of carry bits during a single cycle, instead of one bit at a time.

CONCURRENT ERROR DETECTION

This section deals with definition and evaluation of different architectural solutions to implement fault-tolerant arrays with concurrent error detection. Linear and rectangular array processors are considered to show the flexibility of our approach and the independence from the topology of the interconnections.

To design arrays with error detection capability, two main approaches may be adopted at different architectural levels:

- local error detection verifies *locally* the computation of each processing element,
- global error detection operates at *array level* to identify the error occurrence in the final outputs of the array.

An accurate evaluation of both of these techniques is discussed to provide guidelines for optimal design of application-specific architectures. In particular, we deal with minimisation of the additional silicon area and/or computational delay without reducing the error detection capabilities for an effective WSI implementation.

In *local error detection* the computation is verified locally inside each processing element before delivering results to the adjacent elements. An example is shown in fig. 6 for a rectangular array. In dedicated arrays, local checking may be performed at the output of each computing unit (both logical and arithmetic) or at the output of the whole processing element. In the former case, a checker must be connected to the output of each unit, while in the latter only one checker is necessary for each processing element. Since usually a processing element is composed by more than one computing

unit, the first solution is too expensive in terms of silicon area and computational delay. On the other hand, the error detection capabilities of Berger codes are not affected by the reduced number of checking points in the processing element. If faults are not masked by input data, the difference between the expected and the actual check symbols is preserved by operations which are executed after the error occurrence. The only advantage is minimisation of fault latency.

In semi-programmable architectures local checking may be activated after the completion of each operation or at the end of the computation of the whole processing element. The hardware redundancy is the same in both of these solutions since the processing element must contain one checker. The computational delay is higher in the first structure since more data checks are executed. Also in this case redundancy is not useful since errors are not masked by subsequent faults.

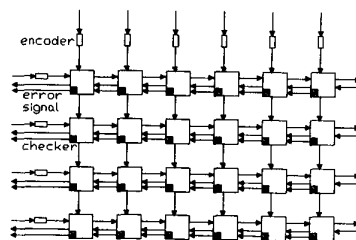


Fig. 6 - Local error detection

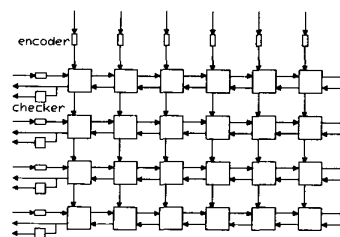


Fig. 7 - Global error detection

In *global error detection* the output results are tested at array borders. Internal processing elements are allowed to deliver their outputs to the adjacent elements without any checking: only the final results of the computation performed by the whole array are verified. A great amount of silicon area and computational delay is saved since only one checker is required for each output data flow. An example is presented in fig. 7.

Computation is lightly delayed with respect to the basic architecture without detection capabilities since additional time is required to generate the redundancy bits. By overlapping computation and final checking it is possible to minimise throughput reduction. Checking can be in fact performed during the computation of the subsequent array output since no decoder is necessary for Berger codes. Silicon area is larger than the area of the basic architecture since there are encoders for the input data, additional circuits to compute the actual check symbols of the results, and checkers to verify the presence of errors.

The use of Berger codes in arithmetic array processors (i.e. in arrays executing only arithmetic operations) is generally more expensive in terms of silicon area and computation delay than the architectural solutions that can be implemented by means of pure arithmetic codes, e.g. $A * N$ and multiresidue codes [5] [6]. Such additional costs are due to the intrinsic difference between the characteristics of the arithmetic and logic operations. The homogeneity of the arithmetic operations allows the definition of efficient coding techniques and compact redundant arithmetic units and checkers. The logical operations are not able to preserve the encoding schemes of classical arithmetic codes: no simple algorithmic relationship may in fact be defined between the coded inputs and the coded output of a logical operation when an arithmetic code has been adopted. The coding scheme of Berger codes is preserved by arithmetic and logic operations and allows simple computation of check bits, but requires a larger redundancy both in time and in silicon area.

A technology-independent evaluation of structures for global error detection is given in fig. 8 for some experimental rectangular architectures. We computed the percentage increase of the silicon area with respect to the basic architecture, the percentage increase computational delay and the percentage decrease of the throughput versus the number of data bits. The silicon area is assumed proportional to the number of logical gates: for simplicity sake, all kinds of gates occupy the same area. The computational delay and the throughput are evaluated by considering the number of gates in the longest path of each computing unit and by assuming that the computational delay is not dependent on the kind of gate.

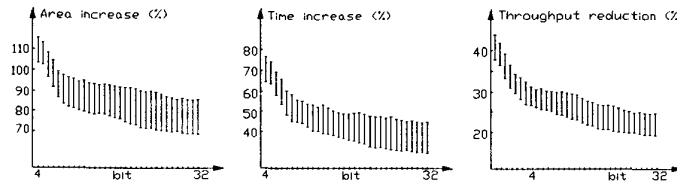


Fig. 8 - Performance evaluation of global detection

CONCURRENT FAULT LOCALISATION

The architectures for error detection are able to assert the presence of an error, but they do not supply enough information to localise the faulty module in the array. Whenever an error is detected, the computation is not reliable until the faulty element is excluded from the computation by reconfiguring the interconnections [3]. Fast fault localisation is important to achieve a high availability of the system since it allows minimisation of the repair time for reconfiguration.

If local error detection is used, the localisation of the faulty element consists simply in observing the error signals generated by the processing elements. In a linear array, the faulty element is the farthest one from the processing element which generates the final outputs of the array. The same approach may be adopted in rectangular arrays in which the results flow in a single direction (e.g. fig. 2). In rectangular arrays in which the result computed by each processing element is used in different directions, the identification requires a more complex reasoning about the position of the faulty element and the distribution of the active error signals. For example, let's the computations flow from the top to the bottom and from the right border to the left one. If all error signals from the row r_i to the bottom and from the column c_j to the left border are active, the fault occurred in the processing element which lies in the crossing of the row r_i and the column c_j .

In global error detection, two techniques may be used: the concurrent and the semi-concurrent approaches. In the *concurrent fault localisation*, the array architecture and computation are modified to generate redundant outputs which allow an immediate identification of the faulty element. The basic idea consists in introducing a redundant computation which flows orthogonally to the nominal computation: the redundant computation uses the results of the nominal computation of each processing element to generate a redundant output. The redundant data are tested as the nominal one. If an error occurs in nominal computation, it influences both nominal and redundant results. From the observation of the outputs, it is possible to deduce the position of the faulty element.

An application of this approach to dedicated arrays is shown in fig. 9. The silicon area is increased with respect to the architecture with detection capability since an additional computing unit is introduced in each processing element and a checker is used for each redundant computation (i.e. each column). Some computational delay is added to generate the redundant computation; such delay may be avoided by an accurate timing of the computation and by overlapping the execution of the redundant computation to the nominal computation of the subsequent results. In linear array processors this solution degenerates in the architecture for local error detection.

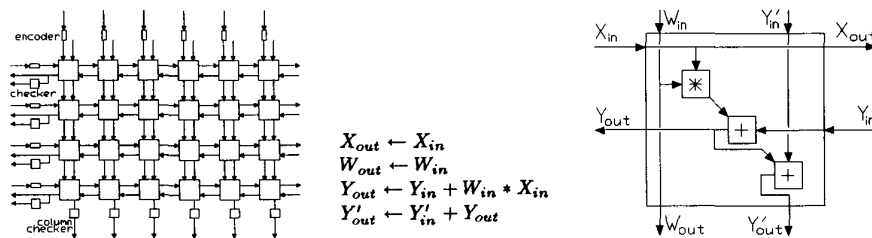


Fig. 9 - Concurrent fault localisation

In semi-programmable arrays we can introduce the redundant computation in the same way presented in fig. 9. Also in this case we can choose, for example, to generate the redundant outputs by adding the nominal outputs of processing elements belonging to the same column. Implementation of this redundant computation may be afforded with one of the following approaches.

- An additional adder for Berger codes is introduced in each processing element and is dedicated to generate the additional output.
- Nominal and redundant computations are completely interleaved. After each operation imposed by the external host of the array for the nominal computation, the redundant computation is executed by using the redundant ALU itself of the processing element. In this case no additional computing unit is required, but only one register to store the partial result of the redundant computation.
- Redundant computation is spread in the nominal computation. After a given number of nominal operations, the host starts the execution of redundant additions. The cost of the hardware is equal to the previous case, while the throughput is greatly improved since operations which are not useful for the nominal computation are reduced. Localisation of the faulty element may be delayed with respect to detection of the error presence.

The choice of one of these approaches is strictly related to the specific application since it involves a detailed balancing of throughput degradation and area increase.

In the *semi-concurrent approach*, the nominal computation based on Berger codes is not modified: localisation is performed by checking one at a time the nominal result computed by each processing element. Information for localisation are generated concurrently with the computation, but checking is performed off-line. An example is shown in fig. 10.

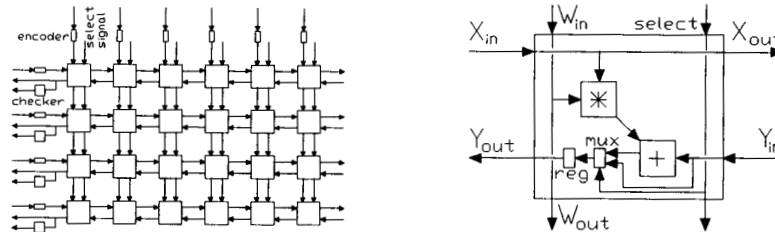


Fig. 10 - Semi-concurrent fault localisation

In this case the nominal computation is suspended during checking. Then, results of processing elements are pipelined and extracted at the array border. Each result is checked as the final results of the nominal computation: in each row of the array, the verification begins from the result of the processing element which is nearest to the array border. In linear arrays, the faulty element corresponds to the last result in which an error is detected during the pipelined checking: if the checker detects j erroneous results starting from the output of the final element which generates the final nominal result of the array, the faulty element is the j -th from that element. In rectangular arrays the identification of the faulty element is performed by checking the results of the row in which the error has been detected.

To implement semi-concurrent fault localisation, a multiplexer and, possibly, a register must be introduced in each processing element. The registers are required whenever the processing element has not latched outputs. The multiplexer is used to implement the pipelined register for extraction of results during the localisation activities and for result routing during normal operation. This hardware redundancy is generally smaller than the corresponding one for concurrent localisation: on the other hand, the localisation time and the repair time are higher.

CONCLUDING REMARKS

In this paper we discussed the use of Berger codes to implement error detection in arithmetic-logic VLSI and WSI array processors. In particular, dedicated and semi-programmable structures composed by parallel computational units are considered.

Concurrent error detection and fault localisation have been shown practically realisable and effective at array level. Unification of checking techniques for arithmetic and logic operations allows a high reduction of hardware redundancy in arithmetic-logic units. On the contrary, the architectures for local fault tolerance should not be used since they are too expensive in terms of silicon area and computational delay.

Further researches should afford detection of larger classes of errors (e.g. errors due to transient or intermittent faults) and efficient correction of errors in arithmetic-logic parallel units. Besides, the extension to programmable array processors should be considered by adopting hardware techniques or coding approaches based upon computation signatures.

REFERENCES

- [1] S.Y. Kung, *VLSI arrays processors*, Prentice Hall, 1988
- [2] J.M. Berger, "A note on error detection codes for asymmetric channels", *Inform. Control*, vol. 4, March 1961
- [3] R. Negrini, M. Sami, R. Stefanelli, *Fault tolerance through reconfiguration in VLSI and WSI arrays*, The MIT Press, NY, 1989
- [4] T.R.N. Rao, *Error coding for arithmetic processors*, Academic Press, NY, 1974
- [5] V. Piuri, "Fault-tolerant systolic arrays: an approach based upon residue arithmetic", *IEEE Proc. ARITH-8*, 1987
- [6] V. Piuri, "Fault-tolerant array processors: an approach based upon $A * N$ codes", *IEEE Proc. ISCAS'88*, 1988
- [7] F.F. Sellers, M.-Y. Hsiao, L.W. Bearnson, *Error detecting logic for digital computers*, McGraw-Hill, NY, 1968
- [8] D.K. Pradhan, S.M. Reddy, "Error-control techniques for logic processors", *IEEE Trans. on Comput.*, 1972
- [9] J.F. Wakerly, *Error detecting codes, self-checking circuits and applications*, Elsevier-North Holland, New York, 1978
- [10] J.-C. Lo, S. Thanawastien, T.R.N. Rao, "Concurrent error detection in arithmetic and logical operations using Berger codes", *IEEE Proc. ARITH9*, 1989
- [11] V. Piuri, "Remarks on the use of Berger codes in array processor design", Int. Rep. 89-048, Dept. of Electronics, Politecnico di Milano, Italy, 1989
- [12] J.C. Lo, S. Thanawastien, "The design of fast totally self-checking Berger code checkers based on Berger code partitioning", *Proc. FTCS 18*, June 1988