Self Checking Systolic FIFO Stack

HUDA B. ABUGHARSA
Higher Institute of Industry
Musrita, LIBYA

ALI H. MAAMAR
Higher Institute of Electronics
Beni Waled, LIBYA

Abstract : The advances in VLSI technology have made possible many changes not only in the amount of hardware that can be integrated into a die permitting the implementation of single chip processor, but also in processor architecture. This creates a need for algorithms that can exploit a high degree of pipelining and parallelism. The algorithms that are the best at this time, for being able to incorporate a high degree of parallelism are the systolic arrays. The systolic systems have balanced uniform architectures which typica11y look like grids where each line indicates a communication path and each intersection represents a ce11 or a systolic element. Unfortunately as the scale of integration has increased so also has the occurrence of intermittent faults. The characteristics of these types of faults render them undetectable by standard test strategies. This is particularly problematic with the wide use of complex circuits in safety-critical applications. Ensuring the reliability of these systems is a major testing challenge. The detection of intermittent faults requires the use of Concurrent Error Detection (coding) techniques. This paper investigates the use of Berger code as a means of incorporating CED into a self checking systolic FIFO stack..

Key-Words : Systolic systems, self checking, intermittent faults, Berger code, Concurrent error detection,

1. Introduction

The advances in semiconductor technology have greatly increased the scale of integration permitting complete systems to be realized as a single chip. This creates a need for algorithms that can exploit a high degree of pipelining and parallelism. The algorithms that are the best at this time, for being able to incorporate a high degree of parallelism are the systolic arrays[1]. Although increased scales of integration offers many advantages, these complex circuits are more susceptible to transient and intermittent faults, a survey [2] [3] [4] has shown that (90%) of hardware related crashes in VLSI systems has been due to these types of faults. With the extensive use of these types of circuits in safety critical application, a major challenge which must be addressed is the development of test techniques to detect transient and intermitted faults. Unfortunately the characteristics of this type of fault, namely random occurrence and short duration, render standard test strategies ineffective. The detection of these types of faults necessitates the use of a test strategy which continuously monitors the operation of the system and compares it with some known reference. This approach is usually referred to as Concurrent Error Detection (CED) that can be achieved through the use of Redundancy. Redundancy is the use of extra resources beyond the requirements of the unchecked system. All CED techniques introduce some form of redundancy. There are three types of redundancy, namely Hardware Redundancy, Time Redundancy, and Information Redundancy. Information redundancy (coding techniques) has been identified as a viable mechanism for implementing concurrent error detection (CED) in VLSI circuits. Invariably, the incorporation of CED schemes incur penalties on a design in terms of area overheads resulting from the additional hardware and routing space necessary to implement the scheme, the area overhead incurred is a function of the number of the check bits (extra bits added to information bits) used in the coding scheme. Amongst all of the separable codes used in CED schemes, Berger code [5] is the least redundant separable code capable of detecting all unidirectional errors. The construction of the code, and its error detection capabilities are discussed below together with a design of a self checking systolic stack using Berger code.

2. Berger Code

Berger code is a separable and unordered code [5][6], it is separable because the information bits and the check bits (check symbol) in the code word are separate, it is an unordered code as it is not possible to change one codeword into another codeword by simply changing either l's to O's or O's to l's, this means that the code can detect all unidirectional errors. The codeword of the Berger code is formed by appending the check bits to the information bits, the check bits of the code is the binary representation of the number of O's (or the complement of the number of 1 's) in the information bits, the number of check bits k = [Log_2 (I + 1)], where I is the number of bits in the information bits (data word), the number of bits in the codeword n = I + k bits. If the number of information bits in a Berger code is I =2 ^{k} -1, k≥ 1, then it is called a maximal length Berger code; otherwise it is known as the non-maximal length Berger code. For example, the Berger code 1100101011 is maximal length because k=3 and I =7= (2 ^{k} -1), whereas 110100011 is non-maximal length because k=3 and I=6 ≠ (2 ^{k} -1).

3. Systolic Systems

The term systolic originated in the medical community where it is used to describe the human circulatory system. Systolic processes, like the circulatory system, perform the operations in a rhythmic, incremental, ce11ular and repetitive manner much like the heart circulating blood through the arteries, veins, and capi11aries. The systolic computation is restricted by the array's operations, much the same way that the heart controls blood flow to the ce11s since it is the source and destination for a11 blood [7]. In 1978 the first systolic arrays were introduced by *Kung* and *Leiserson* [1] as a feasible design for special purpose devices. The systolic systems have balanced uniform architectures which typica11y look like grids where each line indicates a communication path and each intersection represents a ce11 or a systolic element. Systolic arrays are suitable for VLSI implementation because of the following advantages: 1- Their local interconnection schemes avoid the clock skew which arises if data is broadcast over paths of different lengths. 2- The signal drivers are independent of the number of ce11s in the array, and the size of the array can be increased without altering other design parameters. 3- They are easy to implement because of their simple and regular design, one has to design and test only a few cells in order to build a complete system.

4. Systolic Stack

Many programming applications require that a data item be inserted into a set, and at any time it can be popped back out from the set. This type of data structure is ca11ed Last In First Out (LIFO) stack [8]. The device derives its name from the fact that items may be added to the stack in sequential order, and then popped out from the stack in reverse order. There are two fundamentally different architectures for implementing this simple device; a memory stack which is essentially a portion of a memory, this type of stack can grow and may occupy the entire memory space if necessary. The other type of stack is a register stack which is constructed using shift registers the size of the stack is limited by the size of the shift registers, and the operations of the stack are executed without reference to the memory. In this paper we wi11 design and implement a self checking systolic LIFO register stack. Gulbas and Liang [9] have presented a systolic algorithm for the last in first out (LIFO) stack. The stack is an array of ce11s, with each ce11 communicating with its left and right neighborhoods. The cells can either be in the occupied state or the empty state. The ce11s in the systolic stack are fu11 words, each ce11 stores one word. Each word has its own control circuit (state contro11er), the control circuits are the same except the control circuit of the first ce11 (temporary storage ce11) which is connected to the host and receives the push signal, and the control circuit of the second ce11 which also connected to the host and receives the pop signal. All other cells' control circuits are the same, an having neither connections to the host nor they are under the control of the host.

4.1 The Systolic Stack algorithm

The systolic stack design is based on an algorithm which has two rules[9]. *Rule 1:* If there are ever two occupied ce11s to the left of an empty ce11, then the element in the rightmost of the two occupied cells will move one over to the right into the empty ce11, as shown in Fig.1. *Rule 2***:** If there are two empty ce11s to the left of an occupied ce11, then the element in the occupied will move one over to

the left into the rightmost of the two empty ce11s, as shown in Fig.2.

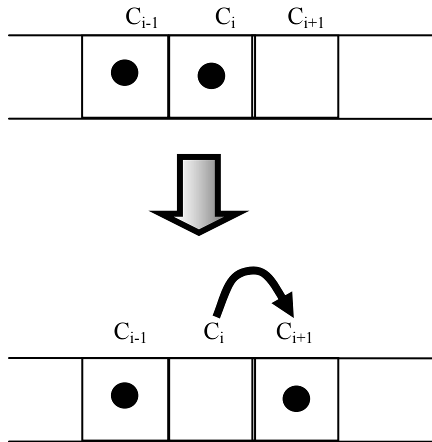

Fig.1 Rule 1 of systolic stack algorithm

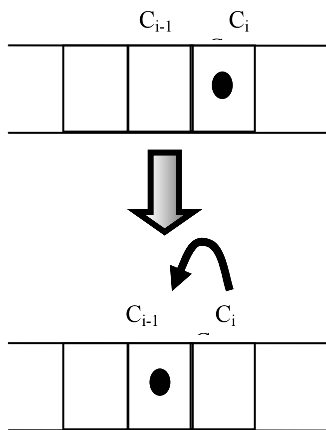

Fig.2 Rule 2 of systolic stack

4.2 Storage array

The storage array is the main circuit of the stack, it consists of an array of shift registers R_1 R_2 ,.... R_n , where n is the number of rows in the array. The shift registers are arranged in rows, where the number of rows determines the number of bits in each word. The number of bits in each shift register, which must all be of the same length, determines the number of words, that can be stored in the stack. Each column of the array is a full word which is controlled by one state controller. The storage array is constructed of shift registers which are arranged in columns. All the shift registers have the same length, build up from the same basic cell, and has two inputs and two outputs. There are four control signals to control the data movement.

4.3 Control circuit

The control circuit in the systolic stack is the most complicated of the whole stack. there is a state controller for each word in the stack. The state controllers of the words pass the information among them selves, each controller communicating with its two most left neighbors as will as its neighbors to its right. There are three types of state controllers: the state controller for the temporary cell (CUo), the state controller for the first cell (CUI) in the stack (top of the stack), and the state controller for the rest of the cells in the stack (CUi), which used n-l times (where n is the number of words that can be stored in the stack), Fig.3.

The heart of each controller is a two J-K master slave flip-flop which are cascaded together. There are some combinational logic gates to the inputs of the first J - K of each controller, the combinational logic is not the same and depends on the controller.

Controller of temporary cell Co (CUo): The function of CUo is to receive the push signal from the host and allow the new word to be pushed into the stack; provided that the stack is not full (at least one cell is empty). As soon as the top of the stack becomes empty, and if there is a word waiting in the temporary cell, then CUo controller will transfer that word waiting to the top of the stack. The controller generates also the signals needed to refresh the contents of the temporary cell. It should be noted that data will stay in temporary cell for short time only, waiting for the rest of the stack to be rearranged.

Controller of the top of the stack (CU_i): The Control circuit of the top of the stack has three functions. First, it should send the word stored in the top of the stack to the data bus, when there is a pop signal from the host. Second, it should transfer the top word of the stack to the left (cell 2) when there is a new push from the host and the stack is not full, the third function is to refresh the contents of the first storage cell (top of the stack) when there is no pop or shift left; that means shifting the data left from the second cell to the first cell when it is empty, and shifting the data right to the second cell, when both the temporary cell and the first cell are full.

Controller of the standard Cell; CU_i is the state controller for the rest of the stack. Each storage cell of the stack has its own state controller; they are the same except for the temporary cell and the top of the stack. The main functions of CU_i are: Move data from C_i to C_{i+1} if C_{i-1} and C_i are occupied and C_{i+1} is empty. Move data from C_i to C_{i-1} if C_{i-1} and C_{i-2} are empty.

4.4 Self Checking Hardware

When designing a circuit which incorporates a concurrent error detection capability, the question immediately arises regarding the number and placement of the checkers; this is trade-off between area and error latency time, that is the delay between the error occurring and its detection. The number of checkers is usually equal to the number of the major buses used to connect the main blocks together of the system. Figure 4 shows the self checking systolic stack, compared with figure 3, it is clear that there is an extra hardware. In self checking stack two check symbol generators (CSG) circuits are needed, one checker (Two Rail Checker), and an extra storage cells (K) attached to each word, these cells are used to store the check symbol generated for each word pushed into the stack, the number of the bits of the check symbol depends on the code used and also on the size of the data word. For example, if the size of the data word that can be pushed into the stack is 32 bits (1=32), and since we used Berger code, then the number of bits of the check symbol is 6 bits (K=6). When data word is pushed into the stack, its check symbol should be generated and pushed into the stack, as the data word moves to the left or to the right in the stack its check symbol should also follows the data word. When data is to be popped out from the stack via the bus, the data should immediately be checked for any detectable errors. This is carried out by the checker, which consists of a check symbol generator for Berger Code, and a Totally Self Checking (TSC) Two-Rail Checker (TRC). The check symbol generator is a zero counter as presented in [10][11], it counts the number of zeros in the information bits of any information word, and gives the number of zeros which represents the check symbol. When the check symbol becomes available it is then compared with the stored check symbol (which generated when the data word pushed in the stack) for that particular word to be popped out, if the stored check symbol and the generated check symbol of the popped word are match then the data word is error free and can be moved out from the stack, but if they not match then the data word is not error free word.

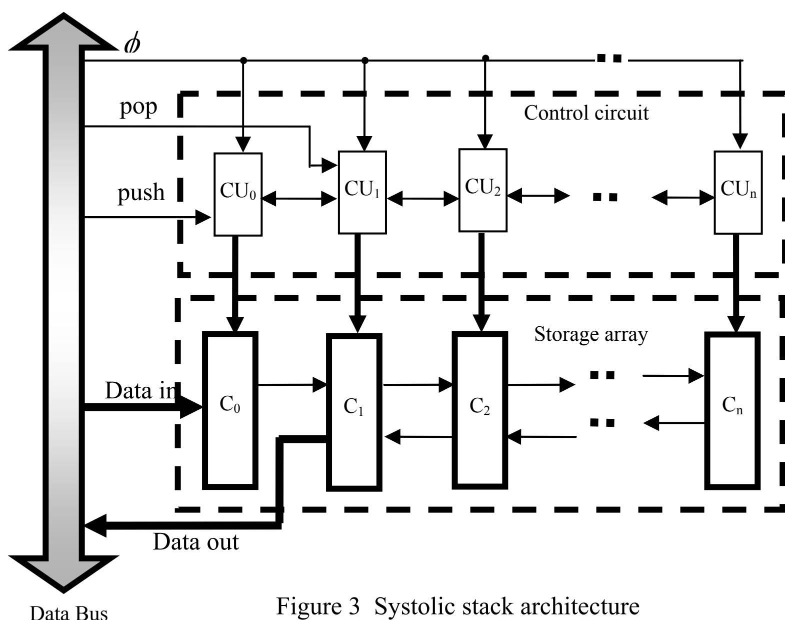

Figure 3 Systolic stack architecture

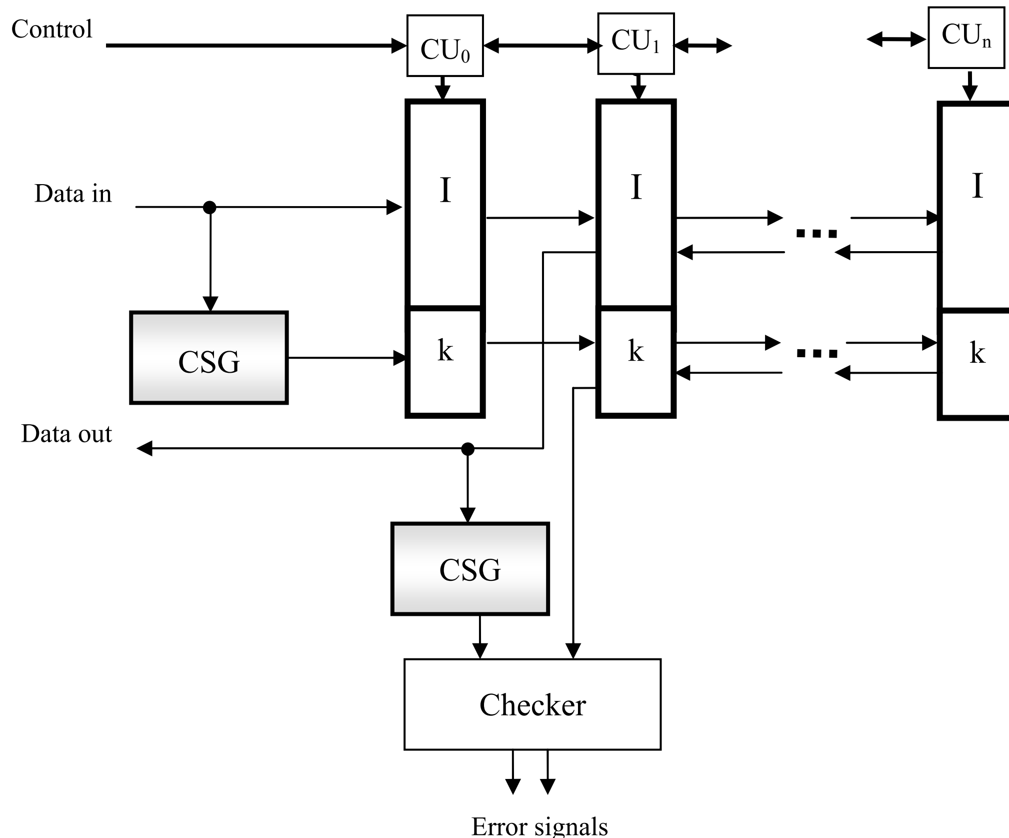

Figure 4 Self checking systolic stack

5. Conclusion

The work in this paper is concerned with the investigation of Berger code as a means of integrating a Concurrent Error Detection (CED) scheme into a VLSI circuit. Berger code has the advantage that it can detect all unidirectional errors The design of a Self-Checking Systolic Stack using Berger code have been presented, the stack is self checking against errors affecting the data word bits and its check bits. Since the code used is a separable code then. The speed of the systolic stack is independent of the capacity of the stack. The systolic stack can be expanded to any desired capacity with out altering the design parameters.

References:

[1] H. T. Kung, C. E. Leiserson, "Systolic Array for VLSI", In lain S. Duss and G. W Stewart, editors, Sparse Matrix Proceeding, 1978.

[2] Parag K. Lala, "Self Checking and Fault tolerance digital Design",Morgan Kaufmann Publisher, 2001.

[3] J. B Clary, R. A. Sacane, "Self-Testing Computers", IEEE computers, Vol. no.10,October 1979, pp. 49-50.

[4] N. K.Jha, S. Gupta, "Testing of Digital Systems", Cambridge University Press 2003.

[5] J.M. Berger, "A Note on Error Detecting Codes for Asymmetric Channels", Information and Control, vol4, pp. 68-73, 1961.

[6] Lala Parag k., "Digital Circuit Testing and Testability", Academic Press, 1997.

[7] D. De Baer. J. Paredaen~ "Parallel Algorithms and Architectures", Springer Berlin / Heidelberg. 1987.

[8] Thomas L. Floyd "Digital Fundamentals" , Prentice Hall, July 2005

[9] L. Guibas and F. Liang, "Systolic stacks, queues, and Counters," in Proc. Conf. Advanced Res. VLSI, MIT, Cambridge, 1982.

[10] S. J. Piestrak, "Design of Encoders and Self-Testing Checkers for some Systematic Unidirectional Error Detecting Codes", the Workshop on Defect and fault-Tolerance in VLSI systems, 1977.

[11] Jerzy W. Greblicki, Stanislaw J. Piestrak ,"Design of Totally Self-Checking Code Disjoint Synchronous Sequential Circuits", Springer Berlin Heidelberg, Feb 2004.