

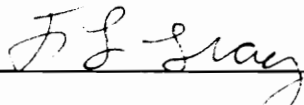
RECOVERY FROM TRANSIENT FAULTS IN WAVEFRONT PROCESSOR ARRAYS

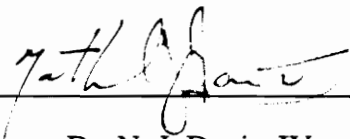
by

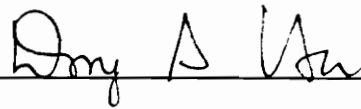
Vinay Murthy

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment for the requirements for the degree of
Master of Science
in
Electrical Engineering

APPROVED



Dr. F. G. Gray, Chairman

Dr. N. J. Davis, IV

Dr. D. S. Ha

July, 1993

Blacksburg, Virginia

C.2

LD
5655
V855
1973
M877
1.2

RECOVERY FROM TRANSIENT FAULTS IN WAVEFRONT PROCESSOR ARRAYS

by

Vinay Murthy

Dr. F. G. Gray, Chairman

Electrical Engineering

ABSTRACT

A transient fault in an array of processing elements results in an inconsistent or incorrect state in the processing element. If the erroneous information has already propagated before detection occurs, then the neighboring processing elements can also be in an incorrect state. Restarting the computation from the beginning every time a transient fault occurs is not only very inefficient but, in real-time computations, may not be possible. This thesis suggests the idea of "rollback" to recover from transient faults. Rollback is done by saving the state of the processing element at different instants of time. When an error is detected, backtracking is done to a consistent state and computation resumes from that state. The rollback algorithm is distributed in nature so that there is no single point of failure in the fault recovery mechanism.

ACKNOWLEDGEMENTS

The author wishes to thank the chairman of his committee, Dr. F. G. Gray, for his guidance, inspiration and support over the past two years. The author wishes to thank Dr. N. J. Davis, IV and Dr. D. S. Ha for their assistance and advice on the thesis.

Special thanks go my parents, Mr. H. N. Ramamurthy and Mrs. Sumana Murthy, my uncles, Mr. Diva Udupa and Dr. S. Udupa for their love, patience and constant support.

TABLE OF CONTENTS

1. INTRODUCTION	1
1.1 SYSTOLIC ARRAYS	1
1.2 WAVEFRONT PROCESSOR ARRAYS.....	2
1.3 FAULT TOLERANCE IN PROCESSOR ARRAY	3
1.4 STATEMENT OF PURPOSE	4
1.5 THESIS OVERVIEW	5
2. WAVEFRONT PROCESSOR ARRAY	7
2.1 COMPARISON TO SYSTOLIC ARRAY	7
2.2 BASIC ARCHITECTURE	8
2.3 CONCEPT OF COMPUTATIONAL WAVEFRONT	10
2.4 PE CONFIGURATION	13
2.5 INTERPROCESSOR COMMUNICATION	15
2.6 APPLICATIONS OF WAVEFRONT PROCESSOR ARRAY	15
2.7 CONCLUSIONS	17
3. ERROR DETECTION SCHEMES	18
3.1 SELF-CHECKING CIRCUITS	20
3.1.1 DEFINITIONS RELATED TO SELF-CHECKING CIRCUITS	20
3.1.2 ERROR DETECTING CODES	22
3.1.3 UNIDIRECTIONAL ERROR DETECTING CODES	25

3.2 SELF-CHECKING CHECKERS	26
3.2.1 TWO-RAIL CHECKERS	26
3.2.2 PARITY CHECKERS	27
3.2.3 m-out-of-n CHECKERS	30
3.2.4 BERGER CHECKERS	34
3.2.5 SELF-CHECKING FUNCTIONAL CIRCUITS	34
3.3 ERROR DETECTION USING TIME REDUNDANCY	35
3.3.1 APPROACH TO ERROR DETECTION USING TIME REDUNDANCY	35
3.3.2 ERROR DETECTION BY RECOMPUTING WITH SHIFTED OPERANDS	39
3.3.3 ERROR DETECTION CAPABILITY OF RESO-1	40
3.4 CONCLUSIONS	41
 4. ROLLBACK AND CHECKPOINTING	 42
4.1 CACHE AIDED ROLLBACK ERROR RECOVERY TECHNIQUE(CARER)	43
4.1.1 PRINCIPLES OF OPERATION	45
4.2 APPLICATION OF MICRO-ROLLBACK TO VLSI RISC PROCESSOR .	46
4.2.1 SUPPORT FOR MICRO ROLLBACK IN A REGISTER FILE	48
4.2.2 MICRO ROLLBACK OF INDIVIDUAL STATE REGISTERS	50
4.3 CONCLUSIONS	54

5.0. RECOVERY FROM TRANSIENT FAULTS IN WAVEFRONT PROCESSOR

ARRAYS 55

5.1 FAULT DETECTION ASSUMPTIONS 56

5.2 ROLLBACK ALGORITHM 57

5.3 HARDWARE OVERHEAD 65

5.4 SIMULATION PROGRAM 68

5.5 CONCLUSIONS 69

6. RESULTS 70

6.1 SKEW AFFECT 71

6.2 VARIATION OF BUFFER SIZE WITH INPUT PERIOD 74

6.3 VARIATION OF BUFFER SIZE WITH STATESAVE PERIOD 78

6.4 SKEW AFFECT AFTER ROLLBACK 81

6.5 CONCLUSIONS 84

7. CONCLUSIONS 85

BIBLIOGRAPHY 88

VITA 92

LIST OF ILLUSTRATIONS

Figure 2.1. Wavefront Array Architecture	9
Figure 2.2. First Iteration of Wavefront	11
Figure 2.3. Second Iteration of Wavefront	12
Figure 2.4. Processing Element Configuration	14
Figure 3.1. Self-Checking Circuit	19
Figure 3.2. Gate Level Design of TRC2	28
Figure 3.3. Design of TRC4 using TRC2	29
Figure 3.4. Odd Parity Checker	31
Figure 3.5. A 2-out of-4 Checker	32
Figure 3.6. Totally Self-Checking Circuit for Berger Codes	33
Figure 3.7. Error Detection with Hardware Redundancy	36
Figure 3.8. Error Detection with Time Redundancy	36
Figure 3.9. Error Detection with Time Redundancy (modified).....	37
Figure 3.10. Two Approaches to Error Detection using Time Redundancy	38
Figure 4.1. Micro Rollback of a Module	49
Figure 4.2. Register File with Support for Micro Rollback	51
Figure 4.3. FIFO Method for Micro Rollback of Individual State Registers	52
Figure 4.4. RAM Method for Micro Rollback of Individual State Registers	53
Figure 5.1. Transmission of "stop propagation" by Faulty Processor	60
Figure 5.2. Transmission of "stop propagation" for East and West Neighbors	61
Figure 5.3. Transmission of "stop propagation" for North and South Neighbors ..	62

Figure 5.4. Transmission of "stop propagation" through the Array	63
Figure 5.5. Processing Element with Hardware Overhead	66
Figure 6.1. Variation of Skew with Input Period	73
Figure. 6.2.Variation of Buffer Size with Statesave Period	76
Figure 6.3. Variation of Buffer Size with Statesave Period	77
Figure 6.4. Variation of Skew with Statesave Period	80
Figure 6.5. Skew Affect before and after Rollback	82

Chapter 1 INTRODUCTION

Recent advances in the field of VLSI have opened new avenues for implementing increasingly sophisticated signal processing and image processing applications. While a major improvement in device speed is foreseen, it is no way comparable to the rate of throughput required by modern real-time signal processing [McC80, SuM77]. In order to achieve such increases in throughput rate, the only effective solution appears to be highly concurrent processing. Array processors such as systolic arrays and wavefront processor arrays are highly concurrent in nature and particularly suited to signal and image processing applications [FoK80, FiK85].

1.1 SYSTOLIC ARRAYS

In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to the main memory. The data is input to the systolic array at the processing elements that form the boundary of the array. The processing elements, through which data passes, form the different stages of a

pipelined implementation [Ku82]. Systolic arrays are usually two-dimensional to make the system highly parallel.

There are many advantages of systolic arrays. The repetitive modular structure of a systolic array makes it ideal for Very Large Scale Integration (VLSI) and Wafer Scale Integration (WSI). The localized communication in a systolic array reduces the propagation delays and power consumption. Perhaps the most important advantage is the ability to decrease the load on the host machine by implementing parallel computations in hardware which were previously done with software.

One of the major disadvantages of a systolic array is the need to provide global clock synchronization to all the processing elements of the array. Clock skew limits the size of the array. Also, a global clock constitutes single point of failure for the entire array. A wavefront processor array is a systolic array without any global clock synchronization.

1.2 WAVEFRONT PROCESSOR ARRAY

A wavefront processor array is characterized by computational wavefronts that propagate through the array. A wavefront in a processor array corresponds to mathematical recursion in an algorithm. The concept of mathematical recursion will be explained later in Chapter 2. Successive pipelining of wavefronts through the array will accomplish the computation of all recursions. A major advantage of the wavefront architecture is that it provides an asynchronous waiting capability, and hence, can cope with timing uncertainties, such as local clocking, random delay in communications, and

fluctuations in computing times. The details of the communication format between the processing elements of the array is discussed in Chapter 2.

1.3 FAULT TOLERANCE IN PROCESSOR ARRAYS

VLSI/WSI technology is very favorable for the implementation of regularly structured systems such as systolic or wavefront processor arrays, but as the level of integration increases, the possibility of a single or multiple fault affecting the array also increases. These faults can occur either during manufacturing or during the operational lifetime of the array. In a non-fault-tolerant array, the failure of a single processing element may cause the entire array to fail [KoD86].

A fault-tolerant processor array is an array of regular and interconnected processing elements having extra processing elements (spares) and hardware for reconfiguration. There are two approaches for reconfiguration— centralized and distributed. In the first approach, there is a global centralized controller that controls the interconnection of processing elements to remove a faulty processing element and replace it by one of the spare processing elements [ChF89, PoB88, SaS86]. The disadvantage of this type of approach is that the reliability of the controller limits the reliability of the array. In a distributed control mechanism, the processing elements or the controllers associated with processing elements control the reconfiguration operation. In this scheme there is no single point of failure affecting the reliability of the whole array [BeB92, Wh91].

Both the reconfiguration schemes mentioned above are permanent in nature. As soon as a fault is detected in any processing element, a reconfiguration algorithm is

invoked to remove the faulty processing element from the processing array. The removal of the faulty processing element from the array is permanent irrespective of whether the fault itself is transient or permanent. Error statistics have shown that transient faults are by far more likely than permanent ones. In fact, field studies have shown that transient faults account for over 90% of the detected faults [SiS82]. This becomes increasingly true as VLSI technologies are pushed to sub-micron levels of fabrication, characterized by very low power signals and by lower immunity to noise. Unfortunately, mission-critical applications often involve noisy environments. If permanent fault reconfiguration is invoked every time a fault is detected in a processing element, then over 90% of the time, a processing element having only a temporary fault is removed from the array.

1.4 STATEMENT OF PURPOSE

The purposes of the research presented herein are threefold. The main purpose is to propose a transient fault recovery scheme for wavefront processor arrays. The transient fault recovery procedure is to be implemented in the array before permanent fault reconfiguration is invoked. The second purpose is to explore the various error detection schemes for transient faults in processor arrays. The third purpose is to investigate the hardware and time overhead introduced due to the transient fault recovery scheme. The balance needed between the hardware overhead and time overhead, for any algorithm implemented in a wavefront processor array is also investigated.

1.5 THESIS OVERVIEW

Chapter 2 provides an overview of the wavefront processor array. The architecture and implementation of a wavefront processor array is presented. Also discussed are the different kinds of algorithms implemented in a wavefront processor array.

Chapter 3 explores the various error detection schemes for transient fault recovery in wavefront processor array systems. The two methods discussed are— error detection by using totally self checking circuits and concurrent error detection in ALU's by recomputing with shifted operands.

Chapter 4 discusses the definitions and terms associated with transient fault recovery. The various transient fault recovery schemes for uniprocessor systems are also surveyed.

Chapter 5 presents the algorithm for recovering from transient faults in wavefront processor arrays. The hardware and time overhead introduced due to transient fault recovery scheme are also discussed. The description of the software simulation model to test the algorithm is also presented.

Chapter 6 summarizes the results of this research. In particular, a methodology for obtaining a balance between hardware overhead and time overhead for any application in a wavefront processor array is presented.

Chapter 7 presents the conclusions relating to the thesis and discusses open questions for future research.

Chapter 2 WAVEFRONT PROCESSOR ARRAY

The supervisory overhead incurred in a general supercomputer often makes them too slow or too expensive for real-time signal and image processing. Array processors provide the throughput necessary for these applications at a low cost – a result of advances in VLSI technology and improvements in the various CAD tools. Array processors maximize the main strength of VLSI – intensive computing capability and yet circumvent its main weakness – restricted communication [MeC80, SuM77].

2.1 COMPARISON TO SYSTOLIC ARRAYS

The data movements in a systolic array are controlled by global timing- reference "beats." As size of the array increases, it becomes more and more difficult to distribute a common clock throughout the array. This is because of clock skew. A wavefront processor array is a modification of a systolic array in that it does not have any global clock reference. Synchronization is achieved by the data flow mechanism between the

processing elements. Hence, the wavefront array combines the pipelining principle of a systolic array with the data flow computing mechanism. Conceptually, the requirement of correct timing in the systolic array is now replaced by a requirement for correct sequencing in the wavefront array [KuL87].

2.2 BASIC ARCHITECTURE

Most signal processing algorithms involve repeated application of relatively simple operations, with a regular and localized form of communication between the processing elements. There is uniform data flow across the array that leads to the concept of computational wavefront. The computational sequence starts with one element and propagates through the processor array, resembling a physical wave phenomenon. A wavefront array architecture is shown in Figure 2.1. Let t be the time taken for execution in each processing element and n be the input period to the array. The movement of the first wavefront across the array is shown by the double line. The second wavefront follows the first wavefront n time units later in each processing element, as illustrated by a single line. In fact, all algorithms that have locality and recursivity will exhibit the computational wavefront phenomenon [KuL87].

In a wavefront architecture, the data transfer between a processing element and its neighbors is by mutual convenience. Whenever data is available, the transmitting processing element informs the receiver, and the receiver accepts the data whenever required. It then communicates with the sender that data has been accepted. This scheme is implemented by a simple handshake that ensures that computational wavefronts

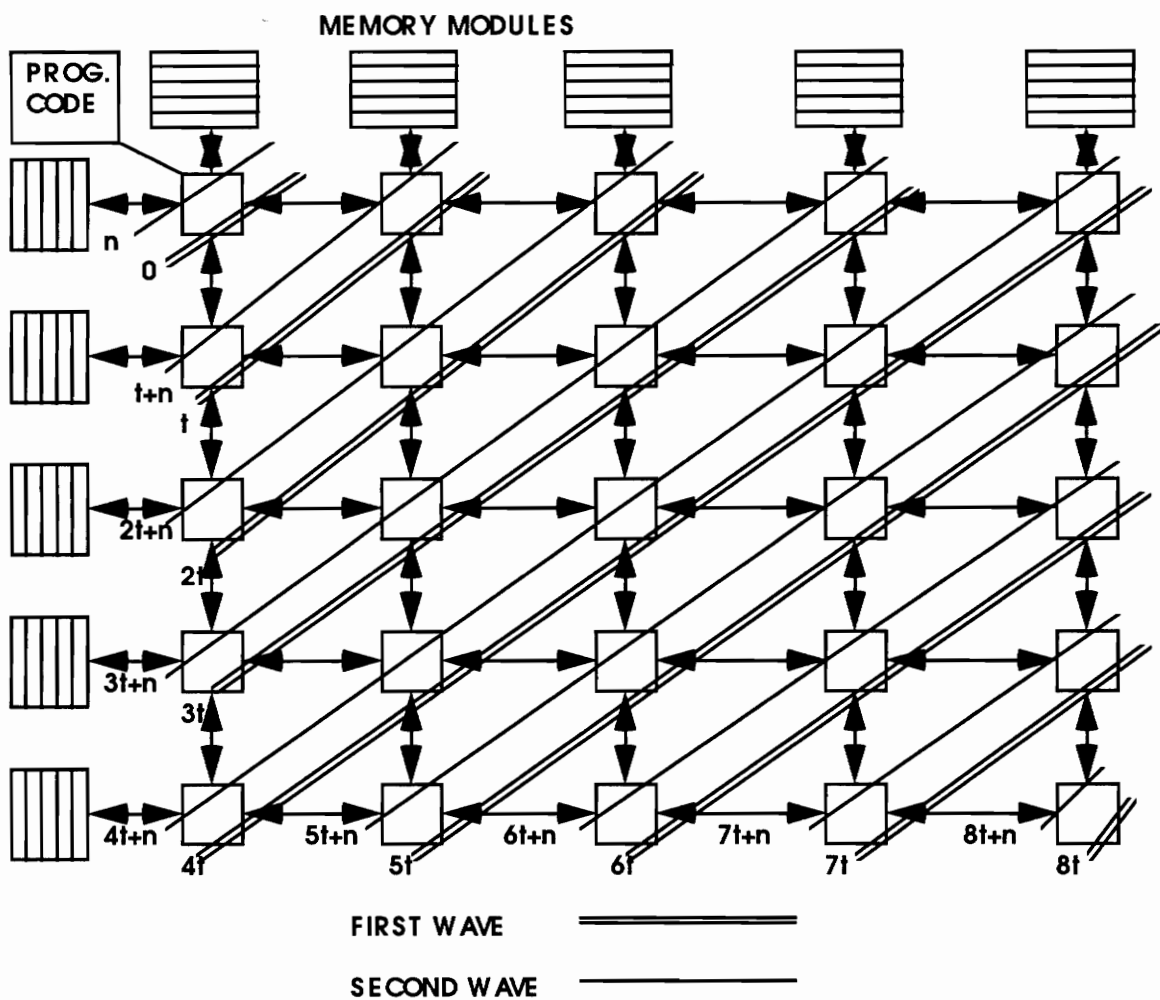


Figure 2.1 Wavefront Array Architecture

propagate in an orderly fashion without crashing into one another. Since there is no need to synchronize the whole array, a wavefront array processor is truly scalable [KuA82].

2.3 CONCEPT OF COMPUTATIONAL WAVEFRONT

Matrix multiplication will be used as an example to explain the concept of a computational wavefront. The wavefront processor array for this example is configured as a square array of $N * N$ processing elements with regular and local interconnections. Let $A = \{a_{ij}\}$, $B = \{b_{ij}\}$, and $C = A * B = \{c_{ij}\}$ all be $N * N$ matrices. If the matrix A can be decomposed into rows A_i and matrix B into columns B_j then the matrix multiplication can be carried out in N recursions by executing

$$C^k = C^{(k-1)} + A_k * B_k$$

recursively for $k = 1, 2, \dots, N$.

For example, consider the computational wavefront for the first recursion in matrix multiplication. The registers of all processing elements are initially set to zero:

$$C_{ij}^{(0)} = 0 \text{ for all } (i,j);$$

the entries of A are stored in memory modules to the left (in columns), and those of B in memory modules on the top (in rows). The process starts with PE(1,1), where

$$C_{11}^{(1)} = C_{11}^{(0)} + a_{11} * b_{11}$$

is computed. The first wavefront for the computation is shown in Figure 2.2. The computing activity then propagates to the neighboring processing elements (1,2) and (2,1), which will execute

$$C_{12}^{(1)} = C_{12}^{(0)} + a_{11} * b_{12}$$

and

$$C_{21}^{(1)} = C_{21}^{(0)} + a_{21} * b_{11}$$

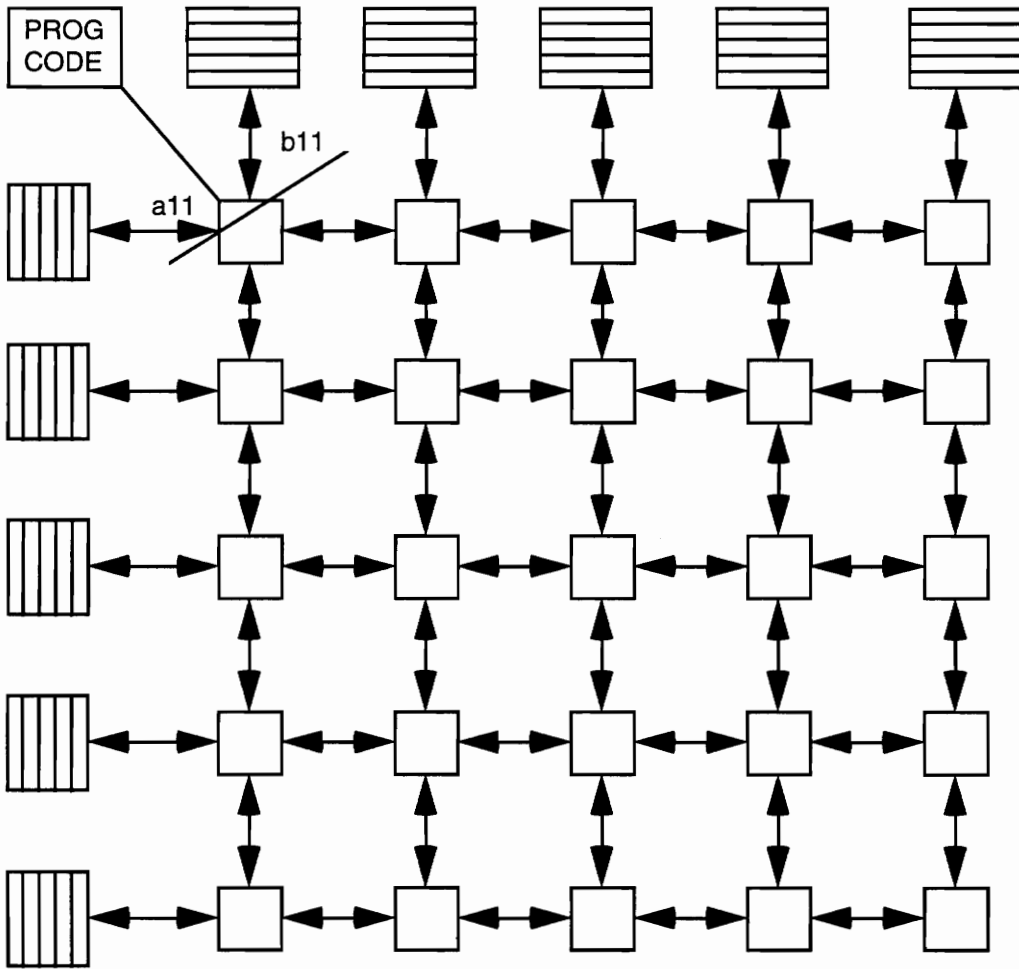


Figure 2.2 First Iteration of Wavefront

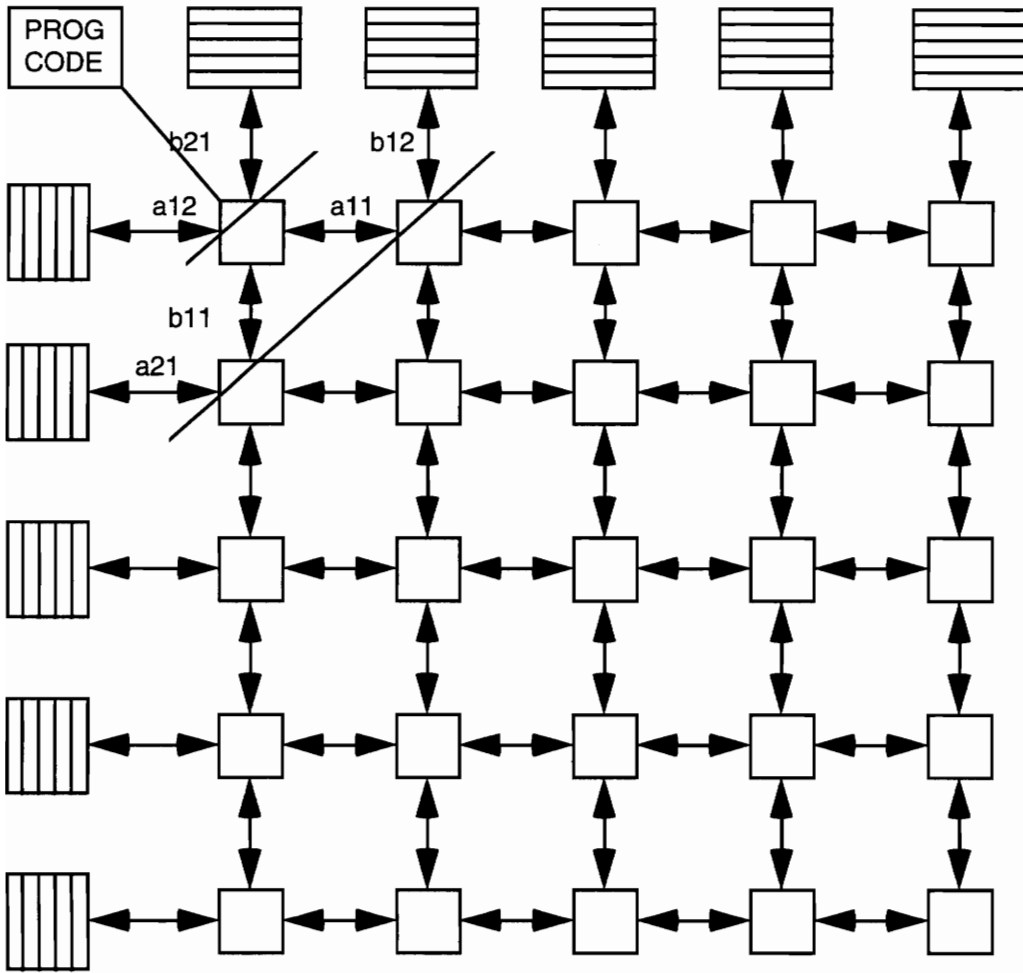


Figure 2.3 Second Iteration of the Wavefront

The wavefront corresponding to the above computation is shown in Figure 2.3. The next front of activity will be at processing elements (3,1), (2,2), and (1,3). Thus, the computational wavefront travels down the array. This computation is similar to the propagation of electromagnetic waves in that each processor acts as a secondary source and is responsible for the propagation of the wavefront. The mathematical recursion is over as soon as the first wavefront sweeps through all the cells. As the first wave propagates, an identical recursion can be executed in parallel by pipelining a second wavefront immediately after the first one. For example, the (1,1) processor will execute:

$$C_{11}^{(2)} = C_{11}^{(1)} + a_{12} * b_{21}$$

and so on. In general, the (i,j)th processor will execute the kth recursion,

$$C_{ij}^{(k)} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + ... + a_{ik} * b_{kj}$$

The pipelining is feasible because the wavefronts of two successive recursions will not intersect [KuA82]. There is no contention because the processors executing the recursions at any given time will be different. All algorithms that possess the property of locality, regularity, recursivity, and concurrency will exhibit the behavior of computational wavefront.

2.4 PE CONFIGURATION

The proposed architecture of the array processing element is shown in Figure 2.4. The design reflects the conventional architecture consisting of an internal program memory, a control unit, an ALU, and a set of registers. The communication of data to the neighboring elements is handled by bi-directional buffers and independent status and control flags for each of its neighbors [KuA82]. The array configuration described upto now has no fault

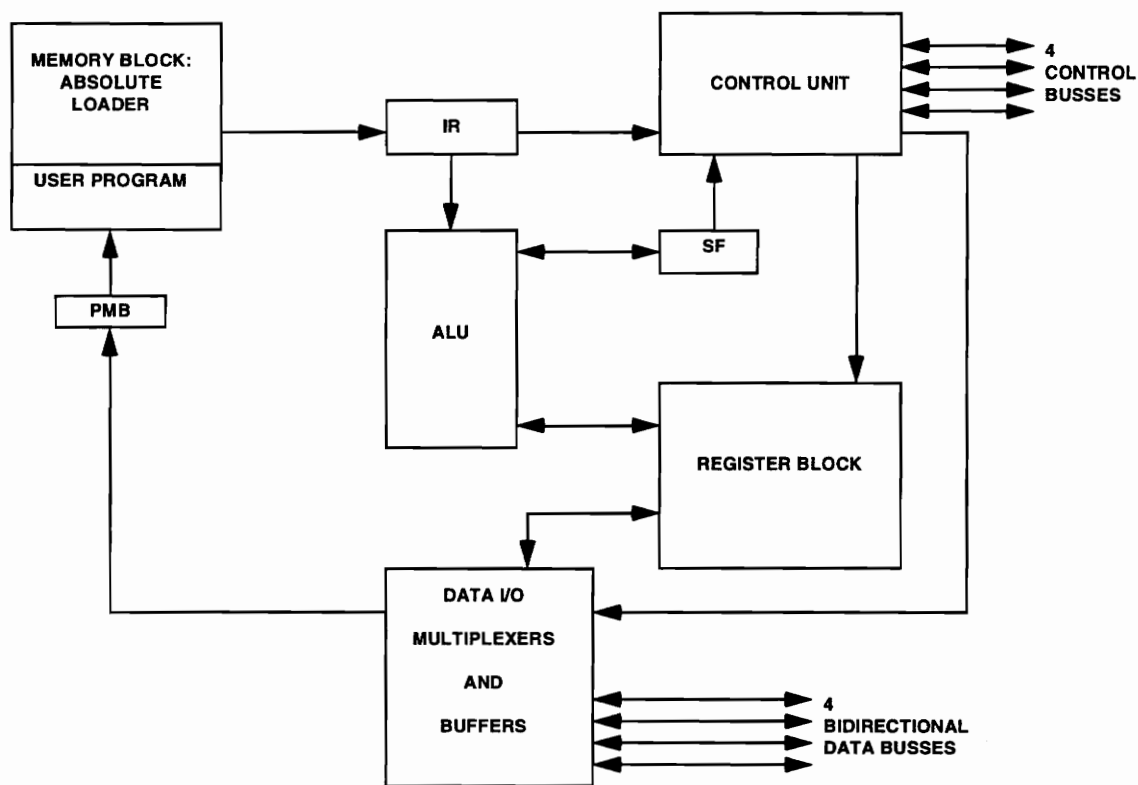


Figure 2.4 Processing Element Configuration

tolerance to recover from transient faults. The processing element modified to have fault tolerance is discussed in Chapter 5.

2.5 INTERPROCESSOR COMMUNICATION

The processing elements of the array wait for a wavefront to arrive, then perform their computation, and finally act as a secondary source of the wavefronts. To implement this wait, the processors are provided with data transfer buffers. Hence, a FETCHing of data involves an inherent WAITing for the buffer to be filled up, DATA READY by the adjacent data sourcing processing element. Any source processor wanting to send data to its neighbor asserts the DATA READY signal. The receiver responds with the DATA USED signal which implies that the receiver has used the data sent previously. To avoid overrunning of data wavefronts, the processor hardware ensures that a processor cannot send new data to the buffer unless old data has been used by the neighbor. Thus, the wavefront concept suggests that interprocessor communication employ buffers and 'DATA READY/DATA USED' flags between processing elements. The WAITS for the wavefronts of data allow for globally asynchronous operation of processing elements.

2.6 APPLICATIONS OF WAVEFRONT PROCESSOR ARRAY

A broad range of applications have been demonstrated on wavefront array architectures [KuL87]. The wavefront principle is applicable to all algorithms that possess the properties of recursivity and locality.

1. Basic Matrix Operations

- a) Matrix Multiplication
- b) LU Decomposition
- c) LU Decomposition with Localized Pivoting
- d) Given's Algorithm
- e) Back Substitution
- f) Null Space Solutions
- g) Matrix Inversion

2. Special Signal Processing Algorithms

- a) Toeplitz System Solver
- b) Linear Convolution
- c) Recursive Filtering

3. Other Algorithms

- a) Partial Difference Equation Solution
- b) Sorting

Recently, a similar computing structure has been applied to parallel processing [KuL87] for speech recognition. It has also been reported that the wavefront concept can be utilized to solve the singular value decomposition (SVD) problem, which is crucial for image-processing and beam-forming applications.

2.7 CONCLUSIONS

The systolic array features the important properties of modularity, regularity, local interconnection, and highly pipelined and highly synchronized multiprocessing. However, the systolic array requires global clock synchronization which is a major barrier towards VLSI implementation. The biggest advantage of the wavefront architecture is that it avoids the problem of global synchronization. It uses the powerful concept of a computational wavefront for synchronization. The processing elements of the wavefront array are programmable which makes them flexible for a variety of applications.

Chapter 3 ERROR DETECTION SCHEMES

Wavefront array processors are highly complex digital systems made by interconnecting many processing elements. The first step in introducing fault-tolerance in wavefront array systems is to have a fault detection scheme. Faults need to be detected as soon as they occur to prevent data contamination. Faults can be of two kinds – transient or permanent. In the era of VLSI circuits, transient faults occur far more frequently than permanent faults, hence detection and recovery from transient faults is very important [SiS82].

In this chapter two methods for detection of transient faults are discussed:

1. Self Checking Circuits
2. Error Detection using Time Redundancy

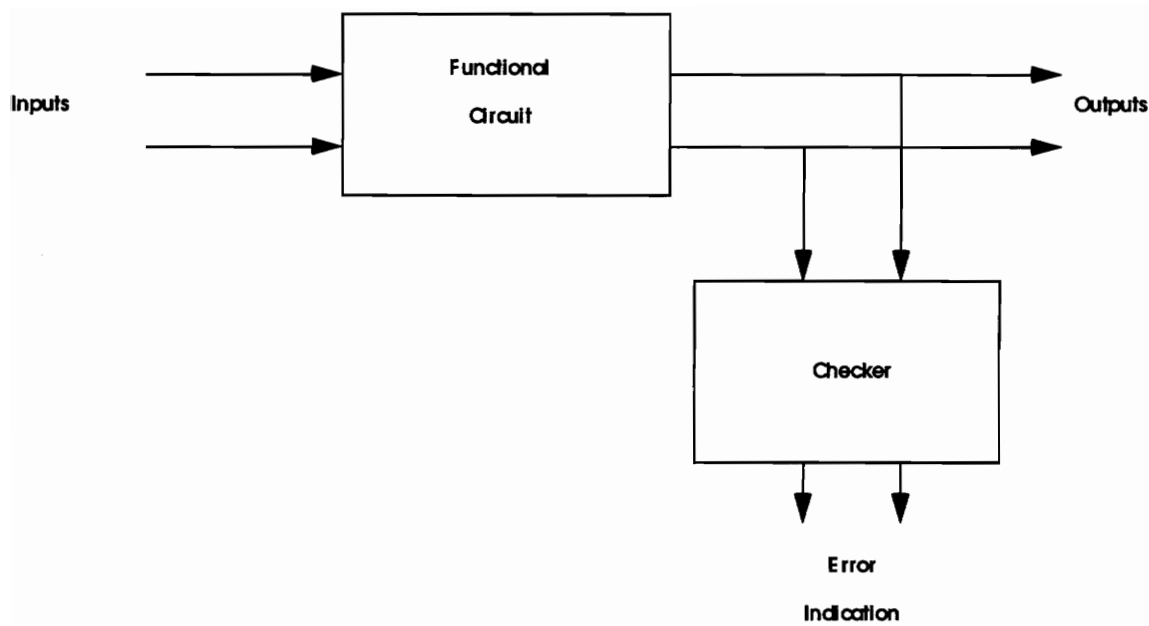


Figure 3.1 Self-Checking Circuit

3.1 SELF-CHECKING CIRCUITS

Self-checking circuits are based on error detecting codes. A self-checking circuit consists of a functional circuit and a checker. The block diagram of a self-checking circuit is shown in Figure 3.1. For self checking purposes, both the inputs and outputs of the functional circuit need to be encoded. For example, if a circuit has p inputs and q outputs, then in all, there are 2^p input vectors and 2^q output vectors. The sets of input and output vectors are each divided into two disjoint sets – one consisting of *codewords* and the other of *non-codewords*. The set of codewords is called the *code-space* and the set of non-codewords is called the *non-code-space*. Under normal fault-free operation, the functional circuit receives a codeword from its input code-space and produces a word from its output code-space. The checker checks to see if the functional circuit has produced a codeword. If a non-codeword is produced then the checker gives an error indication at its outputs.

3.1.1 DEFINITIONS RELATED TO SELF-CHECKING CIRCUITS

The concept of self-checking circuits was first proposed by Carter and Schneider and later formalized by Anderson and Metze [AnM73]. The following definitions are proposed to describe the self-checking property.

Fault-Secure Property: A circuit is *fault-secure* for a set of faults F , if for every fault in F , the circuit never produces an incorrect codeword at the output in response to an input codeword.

Self-Testing Property: A circuit is *self-testing* for a set of faults F , if for every fault in F , the circuit produces a non-codeword at the output for at least one input codeword.

There is a big difference between an incorrect codeword and a non-codeword. If a circuit produces an incorrect codeword, it means that a codeword other than the intended one is produced. Hence, if a circuit were to produce an incorrect codeword due to a fault then the checker would not be able to detect the error. The fault-secure property ensures that all erroneous outputs will be detected. On the other hand, a non-codeword is a word or vector which does not belong to the code-space. In order to expose the faults as they occur, we need the self-testing property.

Totally Self-Checking: A circuit is said to be *totally self-checking* (TSC) if it is both fault secure and self testing.

Code-Disjoint Property: A circuit is *code-disjoint* if it maps codewords (non-codewords) at its inputs to codewords (non-codewords) at its output.

A checker is said to be *totally self-checking* if it is fault secure, self-testing and code-disjoint. Thus, if an error indication is given at the output of the checker, then there is a fault either in the functional circuit or the checker. A self-checking checker has usually two outputs because if it had only one output then, if the output were stuck at the NO ERROR state, the error would never be detected.

TSC circuits work under the following assumptions:

1. Faults occur one at a time.

2. There is a sufficient time interval between successive faults so that all the required codeword tests can be applied to the circuit.

The first assumption is required because the circuit which has been designed to be TSC with respect to single faults may not be TSC if two or more faults are present simultaneously. The second assumption is required to make sure that a fault is detected before the next one occurs. Under these assumptions, a TSC functional circuit always produces a non-codeword as the first erroneous output vector due to a fault. This is known as the *TSC goal*.

3.1.2 ERROR DETECTING CODES

The inputs and outputs of a self-checking system are both encoded with a suitable error detecting code. The choice of code depends on the type of errors that are most likely. The different types of errors can be broadly classified as:

Symmetric: A symmetric error involves an error in one or more bits of the data word, where an affected bit can change from 0 to 1 or 1 to 0.

Asymmetric: When only 0 to 1 or 1 to 0 transitions occur in any data in any data word and the type of transition is known a priori, the error is termed asymmetric.

Unidirectional: A unidirectional error is said to occur if all transitions are either 0 to 1 or 1 to 0, but not both, in any given data word. However, the type of transition is not fixed a priori.

The different types of codes can be classified as:

Systematic Code: In this type of codeword, a check symbol is appended to the information symbol. If the number of information bits is k and the number of check bits is r , then in a systematic code each one of the 2^k information symbols appears as part of some codeword.

Separable Code: This scheme is similar to the systematic code in that the check bits are appended to the information bits, but unlike a systematic code, only $s < 2^k$ information symbols appear as part of the codeword [JhK90].

Non-Systematic Codes: In this scheme, the information bits and the check bits cannot be identified separately.

The advantage of a non-systematic code is that they are less redundant where redundancy is defined as the difference in the number of bits in an encoded data word and a non-encoded data word. The advantage of systematic and separable code is that encoding/decoding and data manipulation can be done in parallel.

Different codes have been proposed for different kinds of applications. Some of the most popular types are discussed below:

Parity code: Let $X = (x_1, x_2, \dots, x_n)$ denote an arbitrary data word that has to be encoded. The check symbol for the parity code consists of only one bit x_{n+1} , which is obtained by

taking the modulo 2 sum of all bits in X . A codeword derived by this scheme will have an even number of 1's hence, it is referred to as the *even parity* scheme. If the checkbit x_{n+1} is obtained by taking the complement of the modulo 2 sum of all bits in X then the scheme is referred to as an *odd parity* scheme. A parity code can detect all single bit errors and any symmetric error with errors in odd number of bits. Since it cannot detect any double errors, the parity code is said to be single error-detecting.

Two-Rail Code: A two-rail code consists of two rails $X = (x_1, x_2, \dots, x_n)$ and $Y = (y_1, y_2, \dots, y_n)$ where $y_i = x_i'$, $1 \leq i \leq n$. Any symmetric error can be detected unless exactly the same set of positions in each rail has errors. A two-rail code can also detect any unidirectional error. If protection against only unidirectional errors is required then there are more efficient codes such as *residue codes*.

Residue Code: The residue Q of a positive number P with respect to divisor m is given by

$$Q = P \text{ modulo } m.$$

The residue code is formed by concatenating the information bits P with the checkbits Q . Thus, it is a systematic code. The divisor m is said to be the checkbase of the code. The number of checkbits r is the smallest integer greater than or equal to $\log_2 m$.

Residue codes are useful in error detection in arithmetic units [Av71]. If P_1 and P_2 are two positive integers whose residues are Q_1 and Q_2 then,

$$(P_1 + P_2) \text{ mod } m = (Q_1 + Q_2) \text{ mod } m$$

$$(P_1 - P_2) \text{ mod } m = (Q_1 - Q_2) \text{ mod } m$$

$$(P_1 * P_2) \text{ mod } m = (Q_1 * Q_2) \text{ mod } m$$

Hence, to detect errors, the result of the arithmetic operation on two integers can be compared with the result of the operation in modulo m on their residues. If m is any odd number greater than 1 then the residue code can detect any single bit error [JhK90].

3.1.3 UNIDIRECTIONAL ERROR DETECTING CODES

The most dominant type of errors in VLSI circuits are unidirectional errors [PrS80]. Codes which can detect all possible unidirectional errors are called *all-unidirectional error-detecting (AUED) codes*.

Let X and Y be two vectors. $N(X,Y)$ is defined to be the number of bits in which X is 1 and Y is 0. For example, if $X = 1000$ and $Y = 0011$ then $N(X,Y) = 1$ and $N(Y,X) = 2$. X and Y are said to be *unordered* if $N(X,Y) \geq 1$ and $N(Y,X) \geq 1$. Otherwise, they are said to be *ordered*. If two vectors are unordered then no unidirectional error can change one vector into another. Generalizing this result, a code C is AUED if and only if every pair of codewords in C is unordered [Pr80, Bo82]. Some of the AUED codes are discussed next.

m-out-of-n code: The m-out-of-n code consists of codewords with n bits, of which exactly m are 1's. Thus, there are $C(n,m) = n!/(n-m)!m!$ codewords in this scheme. The number of 1's in a codeword is called the *weight* of the codeword. Since all codewords have the same weight, this code is also called a *constant-weight code*. However, since these codes are non-systematic in nature, encoding/decoding and data manipulation cannot be done in parallel.

Berger Code: This code is a systematic AUED code [Be61]. There are two ways of obtaining this codeword. In the first method, the check symbol is obtained by counting the number of 0s in the information symbol. In the second method, the check symbol is obtained by taking the complement of every bit in the binary representation of the number of 1s in the information symbol. When there are k bits in the information symbol, the count of 0s or 1s can vary between 0 and k . Thus, the number of checkbits, r , in the check symbol is the smallest integer greater than or equal to $\log_2(k+1)$. It has been shown that a Berger code is optimal among all systematic AUED codes [Fr62]. Due to easy encoding and decoding, this code has found wide popularity.

3.2 SELF-CHECKING CHECKERS

Many methods have been presented for the design of gate-level checkers. In this section, gate-level implementation of checkers of some of the codes mentioned earlier will be discussed.

3.2.1 TWO-RAIL CHECKERS

The n -variable two-rail checker has two rails of inputs $(x_1, x_2, \dots, x_n)$ and $(y_1, y_2, \dots, y_n)$ and two outputs (z_1, z_2) . It is generally designed as a tree in which the nodes are any p -variable two-rail checker, $p < n$ [An71]. The most common choice is to use $p=2$. A p -variable two-rail checker is denoted by TRC_p . The gate-level design of TRC_2 is shown in Figure 3.2. The two variable two-rail code has only four codewords: $(x_1, y_1, x_2, y_2) = 0101, 0110, 1001$ and 1010 . For these codewords the outputs (z_1, z_2) of the checker are $(0,1)$ or $(1,0)$ which can be thought of as a 1-out of-2 code. All other

vectors result in (0,0) or (1,1) at the outputs. Thus TRC_2 is code-disjoint. The four codewords also detect all single stuck-at faults in the circuit. Thus it is *self-testing* for single stuck at faults. Also no single fault can change a (0,1) output to (1,0) or *vice versa*, when a codeword is input to the circuit. Thus the checker is *fault secure*. Since the checker is both self-testing and fault-secure, it is *totally self-checking*.

A design of TRC_4 using TRC_2 is shown in Figure 3.3. It was shown in [An71] that in the design of TRC_n , if the biggest building block in the tree is TRC_p , then a test set consisting of $2p$ codewords can be found which detects all single and unidirectional stuck-at faults in TRC_n . Thus if only TRC_2 is used as a building block then only four codewords are needed to test TRC_n irrespective of the size of the tree.

3.2.2 PARITY CHECKERS

Self-checking parity checkers can be designed for both odd-parity and even-parity codes [CaSc68]. Let $(x_1, x_2, \dots, x_{n+1})$ denote the inputs of the checker, where x_{n+1} is the parity bit. The inputs are divided into two disjoint groups with each group having at least one input. Two parity circuits are then realized, one for each group. A parity circuit can be implemented as a tree of EX-OR gates. For an odd-parity code, the two parity circuits constitute a checker. But for an even-parity checker, the output of one of the parity circuits is inverted.

An odd-parity checker with seven inputs is shown in Figure 3.4 . A single fault in the checker can affect only one of the outputs hence the checker is fault secure relative to

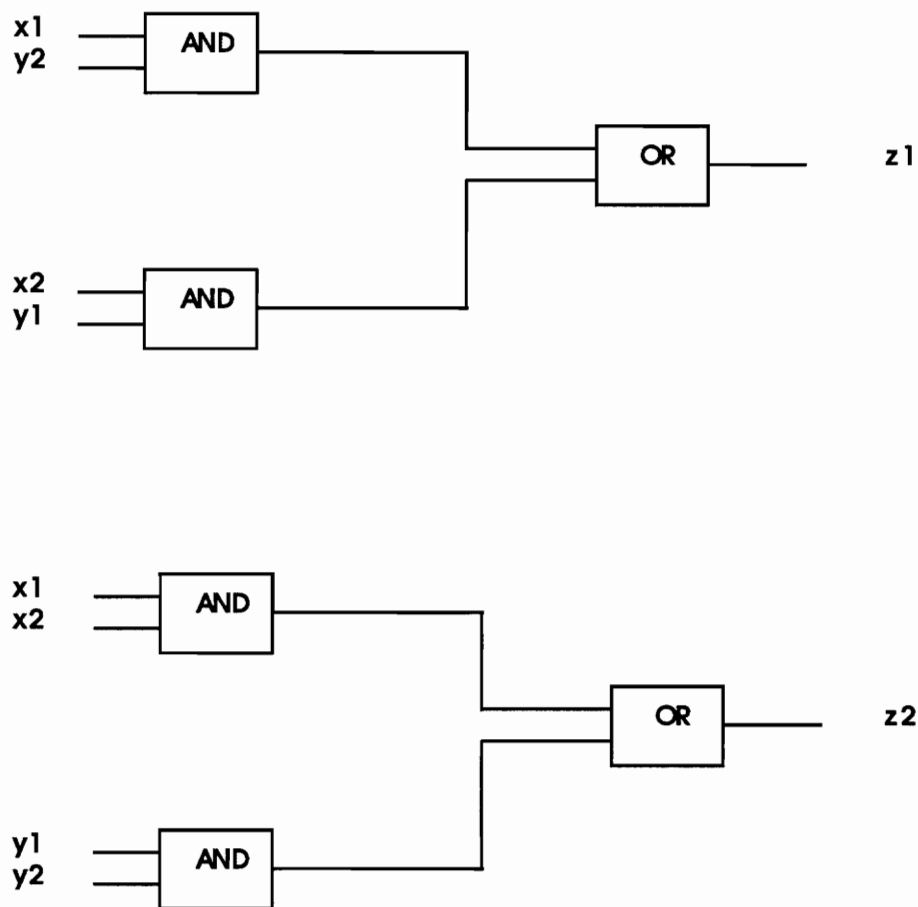


Figure 3.2 Gate Level Design of TRC2

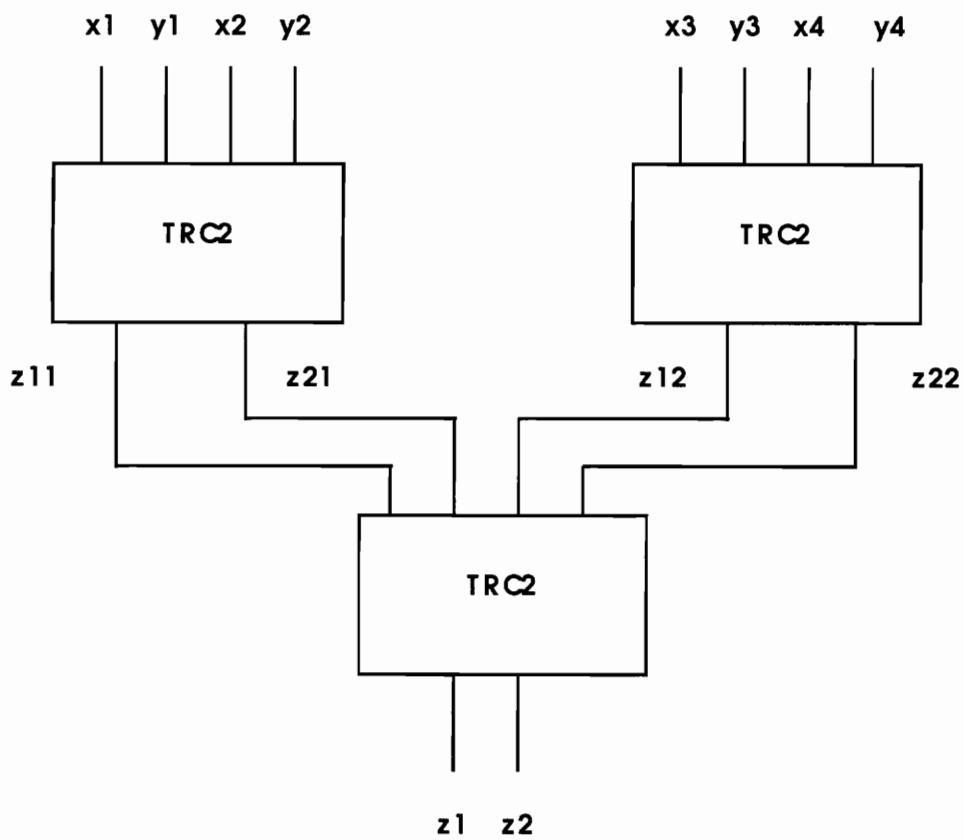


Figure 3.3 Design of TRC4 using TRC2

single faults of any kind. A parity checker in which the largest EX-OR gate has p inputs can be tested by $2p$ codewords [An71]. Hence, for the parity checker shown in Figure 3.4, only four codeword tests are required. One possible set of tests is {0000001, 1110110, 1011101, 0101010}. Therefore, the checker is self-testing for all single stuck-at faults. Finally, the checker is code-disjoint [JhK90]. Hence, it is totally self-checking.

3.2.3 m-out-of-n CHECKERS

Many gate-level totally self-checking checker designs have been presented for m -out-of- n codes [AnM73, Re74, GaH83]. The first design of an m -out-of- $2m$ totally self-checking checker was presented in [AnM73]. Let the $2m$ bits be divided into two groups $A = \{x_1, x_2, \dots, x_n\}$ and $B = \{x_{m+1}, x_{m+2}, \dots, x_{2m}\}$, and let m_a and m_b denote the number of 1s in groups A and B respectively. The two output functions of the checker are given as

$$z_1 = \sum T(m_a \geq i) \cdot T(m_b \geq m-i) \quad i \text{ odd}$$

$$z_2 = \sum T(m_a \geq i) \cdot T(m_b \geq m-i) \quad i \text{ even}$$

where $T(m_a \geq i)$ is a threshold function which is 1 if and only if m_a is greater than or equal to i and "." is the AND operator. The function $T(m_b \geq m-i)$ has a similar meaning.

For example, in the design of a checker for a 2-out-of-4 code,

$$\begin{aligned} z_1 &= T(m_a \geq 1) \cdot T(m_b \geq 1) \\ &= (x_1 + x_2) \cdot (x_3 + x_4) \end{aligned}$$

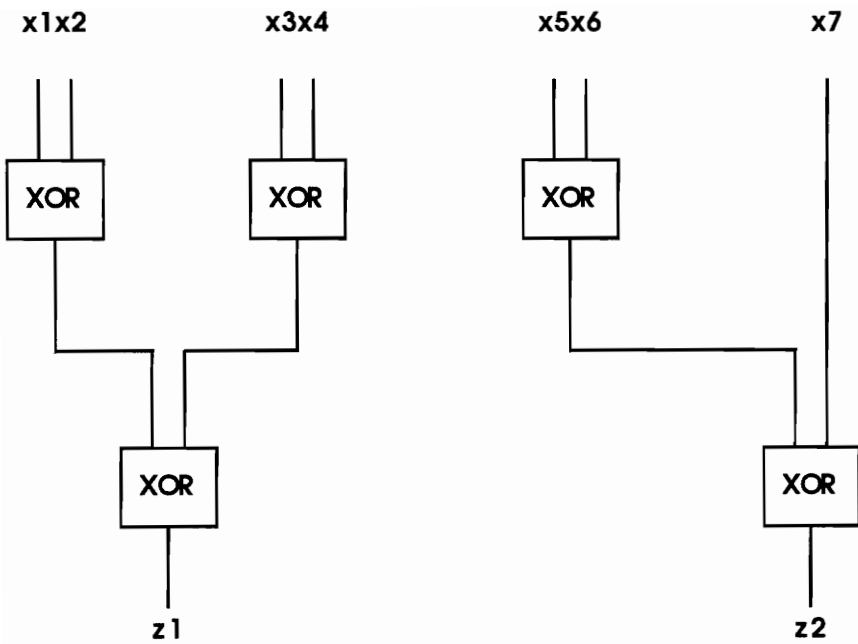


Figure 3.4 Odd Parity Checker

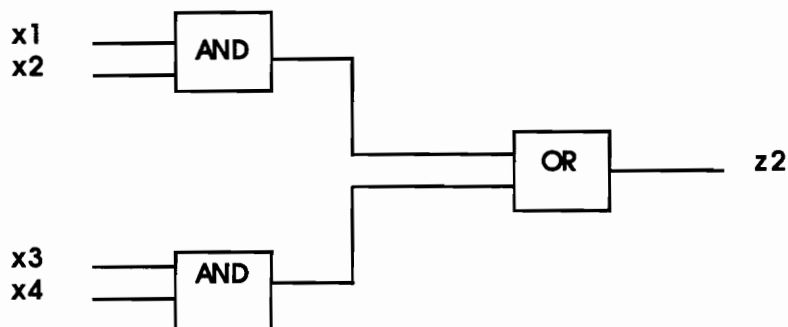
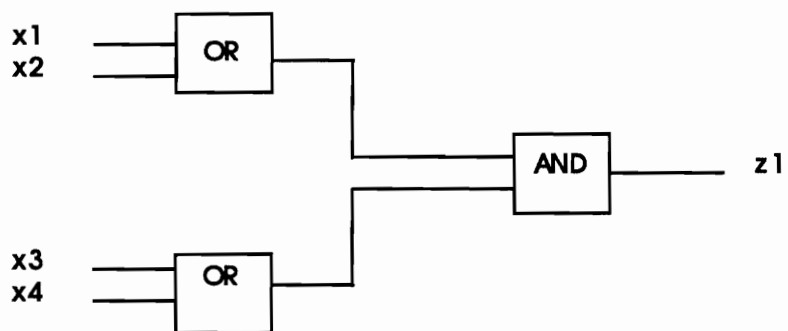


Figure 3.5 A 2-out-of-4 checker

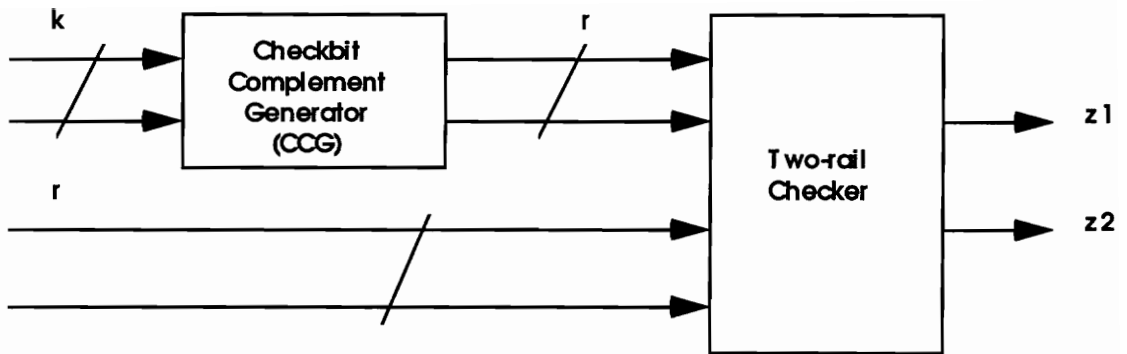


Figure 3.6 Totally Self-Checking Checker for Berger Codes

$$\begin{aligned}
z_2 &= T(m_a \geq 0) \cdot T(m_b \geq 2) + T(m_a \geq 2) \cdot T(m_b \geq 0) \\
&= 1 \cdot x_3x_4 + x_1x_2 \cdot 1 \\
&= x_1x_2 + x_3x_4
\end{aligned}$$

The gate-level design is shown in Figure 3.5. The circuit is totally self-checking with respect to all stuck-at-faults [JhK90].

3.2.4 BERGER CHECKERS

The Berger code is an optimal all-unidirectional error detecting systematic code. The most common totally self-checking checker design for Berger codes is shown in Figure 3.6 [AsR77, MaF78]. The k information bits are fed to a checkbit complement generator (CCG) which generates the bitwise complement of the checkbits. The two-rail checker verifies if the output bits of the CCG are in fact the complement of the r received checkbits. If the CCG is irredundant and the two-rail checker receives all its tests, then the checker is totally self-checking [JhK90].

3.2.5 SELF-CHECKING FUNCTIONAL CIRCUITS

The design of self-checking functional circuits was presented in [SmM77]. In [SmM78], it was shown how strongly fault-secure functional circuits can be designed. A comprehensive theory for the design of self-checking systems was presented in [SmL83].

3.3 ERROR DETECTION USING TIME REDUNDANCY

There are no low-cost and efficient techniques that can do error detection for both arithmetic and logic operations [PaF82]. The error detecting codes such as the Residue code and Inverse Residue code are useful for checking only arithmetic operations. However, these methods are not useful for checking logical operations. Utilizing a fully duplicated logical unit has been recognized as the most effective method for detecting errors in logical operations. For example, the fault-tolerant STAR computer uses inverse residue codes to check the arithmetic unit but duplication for the logic unit [Av71]. In the next section, approaches to devising error detection schemes based on time redundancy are discussed and later a specific method called recomputing with shifted operands (RESO) is discussed.

3.3.1 APPROACH TO ERROR DETECTION USING TIME REDUNDANCY

In this section, a systematic method of error detection using time redundancy is discussed. Let x be the input to a computation unit f and let $f(x)$ be the desired output. A hardware redundancy scheme shown in Figure 3.7, with two identical functions, will detect any error in any one of the computation units. Now, if the same functional unit is used twice in time, as illustrated in Figure 3.8, the result of the first computation step is stored in a register and then compared with the result of the second computation step. A transient fault occurring during either of the two computation steps, but not both will be detected. This method however does not detect any permanent faults.

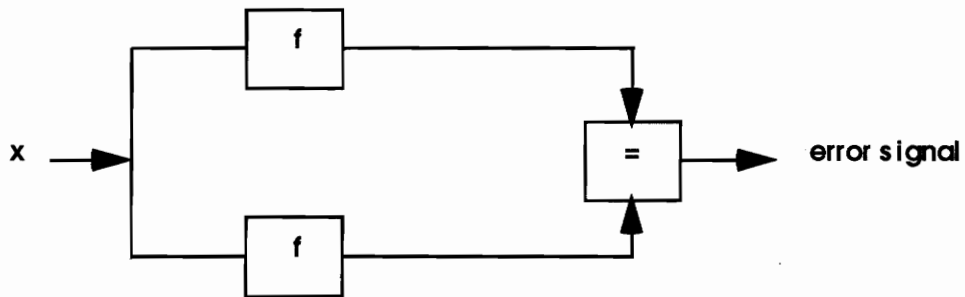


Figure 3.7 Error Detection with Hardware Redundancy

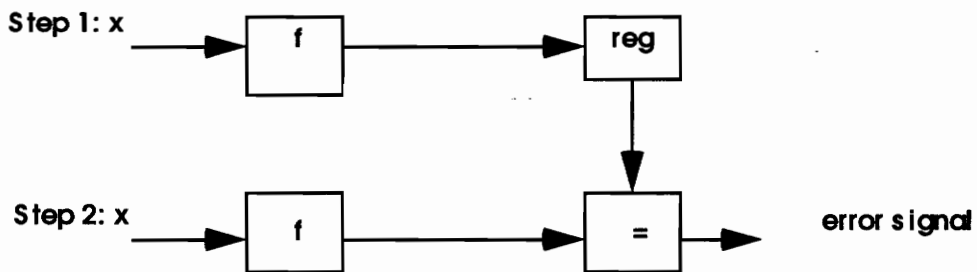


Figure 3.8 Error Detection with Time Redundancy

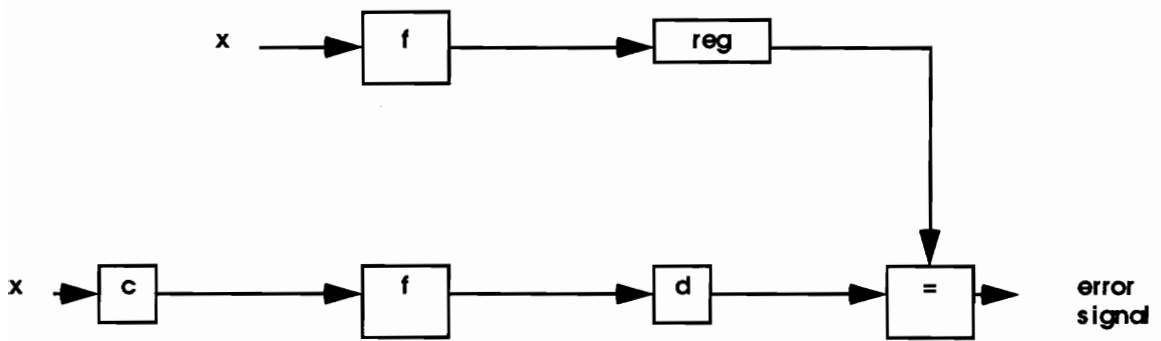


Figure 3.9 Error Detection with Time Redundancy (modified)

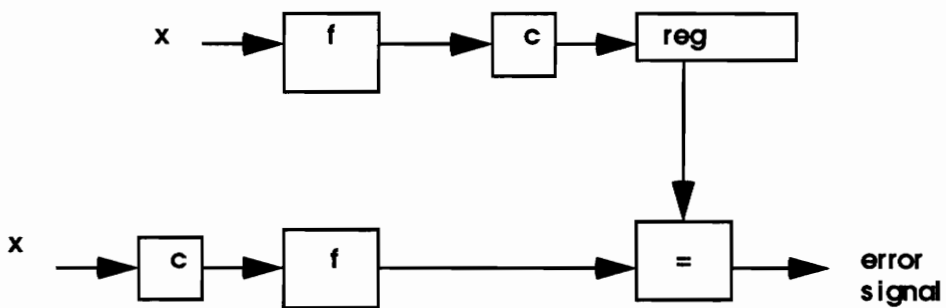
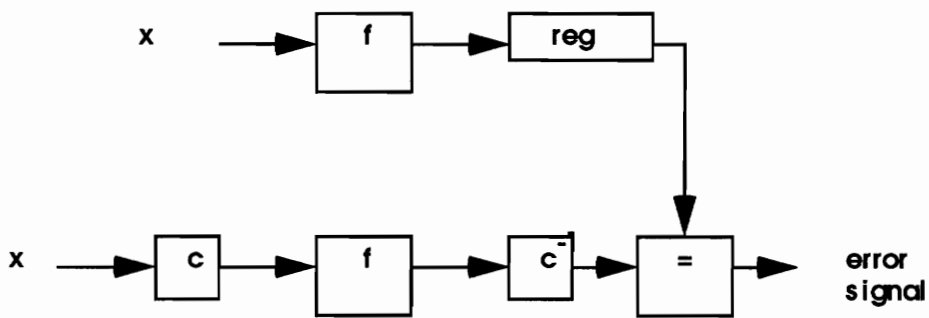


Figure 3.10 Two Approaches to Error Detection using Time Redundancy

The second step of Figure 3.8 can be modified to have two functions c and d such that $d(f(c(x))) = f(x)$ for all x . The value $f(x)$ is computed in the first step and during the second step $d(f(c(x)))$ is computed as shown in Figure 3.9. The functions c and d are called coding and decoding functions respectively. If c and d are properly chosen, then a fault in the functional unit f will affect $f(x)$ and $c(f(x))$ differently. Figures 3.9 and 3.10 are the basis of most error detection schemes using time redundancy.

As an example of Figure 3.10, consider functions $f(x)$ which are self-duals. Let $x = (x_1, x_2, \dots, x_n)$ be an n -bit vector; $f(x)$ is a self-dual if for all x , $f(x_1', x_2', \dots, x_n') = f(x_1, x_2, \dots, x_n)$. Let c be a function which complements each bit of a vector, e.g., $c(x_1, x_2, \dots, x_n) = (x_1', x_2', \dots, x_n')$. Now $c = c^{-1}$ and $c(f(c(x))) = f(x)$. This property of self-duals was the basis of one of the first error detection methods using time-redundancy [ReM78].

There are several factors to be considered in the design of error detection methods using time redundancy. First, a function c has to be found for a given function f such that $c^{-1}(f(c(x))) = f(x)$. The second problem is that the mere existence of the function c does not provide the desired error detection capability. Error detection capabilities vary widely with different c 's. Lastly, the hardware implementation of c and c^{-1} should be simple compared to the implementation of function unit f ; otherwise the hardware redundancy scheme using duplicate functional units will be more cost-effective.

3.3.2 ERROR DETECTION BY RECOMPUTING WITH SHIFTED OPERANDS

Recomputing with Shifted operands (RESO) method is based on Figure 3.10. If f is the functional unit, the coding function c is the left-shift operation, and c^{-1} is the right-

shift operation, then $c(x) = \text{left shift of } x$ and $c^{-1}(x) = \text{right shift of } x$. With a more precise description of c , e.g., logical shift or arithmetic shift by how many bits, what gets shifted in and so forth, it can be shown that for typical ALU operations, $c^{-1}(f(c(x))) = f(x)$ [PaF82].

Using the principle of Figure 3.10, during the initial computation step, the operands are passed through the shifter unshifted and are input to the ALU. The result is then stored unshifted in a register. During the recomputation step, the operands are shifted left and input to the ALU. The result is right shifted and compared with the contents of the register. A mismatch indicates an error in the computation.

When an n -bit operand is shifted left by k -bits, its left most k -bits move out. To preserve these bits during computation, an $(n+k)$ -bit ALU is needed. The leftmost k bits of the $(n+k)$ -bit operands for the first computation step and the rightmost k bits for the recomputation step are determined depending on the operations and the number system.

Let RESO- k be the name of the error detection scheme achieved by recomputing with k -bit shifted operands. The determination of k depends on three factors: implementation of f , the space redundancy allowed, and the set of errors defined. The hardware that the RESO- k needs to detect errors in an ALU for n -bit operations include two shifters, a register, an $(n+k)$ -bit ALU, and an equality checker.

3.3.3 ERROR DETECTION CAPABILITY OF RESO-1

RESO-1 refers to recomputation with operands shifted by one bit. The theorems are stated without proof, the detailed proofs can be found in [PaF82].

Theorem 1: RESO-1 detects all errors in an ALU for all bitwise operations AND, OR, NOT, and their derivations (e.g., XOR, NOR, etc.) when the fault is confined to a single bit-slice.

Theorem 2: In a bit-sliced ALU whose sum and carry functions of the adders are represented by two disjoint networks, any errors will be detected by RESO-1 if the failure is confined to either the sum network or the carry network, but not both.

RESO-1 is incapable of detecting errors if a fault is present in more than one bit-slice. In that case the implementation of RESO-k with $k > 1$ is needed.

3.4 CONCLUSIONS

The self-checking scheme discussed earlier used hardware redundancy for detection of transient and permanent faults. The choice of the error detecting code depends on the type of error that is most likely to affect the operation of the circuit. In a self-checking circuit, the first erroneous data produced is a non-codeword, so there is no propagation of incorrect data. RESO-k uses time redundancy for error detection. The hardware overhead in this scheme is minimum. The disadvantage of this scheme is that the extra time in each processing element for error detection can result in a large time overhead in the overall computation.

Chapter 4 ROLLBACK AND CHECKPOINTING

As VLSI technology is pushed to higher device densities, characterized by very low voltages and power signals, the possibility of occurrence of transient faults has increased dramatically. Therefore, architectures capable of recovering from transient faults have become very attractive. The first step in the transient fault recovery is to detect the transient fault. In Chapter 3, two methods for detecting a transient fault were discussed – self checking and time redundancy.

After a transient fault has occurred, a processor is in an incorrect state. To recover from the faulty condition, a processor has to go back in time to a state that existed before the fault occurred. This process is known as *rollback*. The fundamental assumption of rollback is that the state of the processor is stored at some "safe" place. A processor saves its state at various points of execution called *checkpoints*. Hence, to recover from a faulty state, a processor rolls back to a previous checkpoint and restarts the computation from the saved state.

In the next two sections, rollback mechanisms for uniprocessors, implemented in different hardware systems will be discussed [HuM87, YuT88].

4.1 CACHE AIDED ROLLBACK ERROR RECOVERY TECHNIQUE (CARER)

This technique uses a rollback mechanism in which all error management operations are handled by hardware in a specially designed cache system. The basic idea is to save the state of a CPU at periodic intervals, and in the event of error, restart the computation from the last saved checkpoint [HuM87].

In most computer systems, cache has been "architecturally invisible" i.e., the presence of cache is not reflected in any machine level instructions. When the CPU issues a memory request, the cache fills the request if it can. Otherwise, the request is sent to the main memory. Thus if a write-into cache is assumed, then the cache and the main memory need not be in agreement as to the contents of any given memory location. However, the information stored in the cache is always the current information.

One of the fundamental assumptions of any rollback mechanism is that the rollback state is stored in some "safe" storage i.e., it is inherently more reliable than the system components which are covered by the rollback mechanism.

Some definitions that will help in the discussion of CARER are presented:

Visible Memory: refers to the memory locations which are addressed, either in the main memory or in cache, when the CPU issues a memory request.

Rollback Point: is a point in computation at which information about the state of computation is stored so that it may be used later to recover from an error by restarting the computation from that point.

Rollback Interval: is the period of time between the establishment of two successive rollback points.

Clean Line: is a cache line which has been loaded from main memory but has not been written by the CPU.

Dirty Line: is a cache line which has been written into by the CPU during the current rollback interval.

Unchangeable Line: is a line which has been written into by the CPU during the previous rollback interval. It is called an unchangeable line because it may be read, but not written into by the CPU until it has been copied back to main memory. When it is copied back to the main memory it becomes a clean line.

Invalid Line: is a cache line which contains invalid information. When the system first comes up, all cache locations are automatically marked invalid because no locations have yet been loaded with valid information from main memory.

4.1.1 PRINCIPLES OF OPERATION

When the computer is first brought up, all the cache entries are marked as invalid, and many cache misses are generated for the first few cycles. Each time a cache miss occurs, the line containing the appropriate data is copied from the main memory into the cache, just as in a conventional cache. However, when the CPU wishes to write into main memory, the data is written only to cache and never into main memory, and the entry in the cache is marked as modified or "dirty."

As the cache begins to fill up, it becomes necessary to delete some cache lines to make room for some new data. Clean lines are selected to be disposed of first. Since these lines contain the same data as the corresponding lines in main memory, it is not necessary to copy them back to main memory. Eventually, the cache becomes sufficiently filled with modified lines that it becomes necessary to delete some of the dirty lines to make room for new data. At this point, a rollback point must be established, otherwise any cache miss will result in dirty line being overwritten by some other line.

A rollback point is established by saving the CPU state in internal backup registers, copying all of the dirty cache entries into the main memory, and marking these entries as clean. Execution then continues. If an error is detected, the stored rollback state of the processor is reloaded, and all of the cache lines are marked as invalid.

The major disadvantage with this scheme is that the processor is idle while the dirty cache lines are being copied back to the main memory. This problem is solved by creating another status to identify cache lines which need to be written back to main

memory and which belong to the rollback state. Cache lines which fall into this category are called "unchangeable." Since unchangeable lines belong to the previous rollback interval they must be protected during the current operation.

After the "unchangeable" status has been included, to establish a rollback point, the CPU state is copied and all locations in the cache which have been marked dirty are marked unchangeable. Execution again resumes, but from now on entries in cache marked as unchangeable are written back to the main memory whenever it is necessary to make room in the cache for a new line. Locations marked as dirty are not written back, although they may be modified again by the CPU. Locations marked as unchangeable may be read, but they may not be modified. If a write request is generated for a line that is marked as unchangeable, then that line is first copied to the main memory and then it may be written into.

A rollback point is established by the hardware under the following three circumstances:

- i. When the cache is filled beyond a pre established cache occupancy threshold.
- ii. When an external interrupt occurs.
- iii. When an I/O instruction is executed.

4.2 APPLICATION OF MICRO-ROLLBACK TO VLSI RISC PROCESSOR

Highly reliable systems often require the ability to detect errors as soon as they occur and prevent the spread of erroneous information throughout the system. In order to meet these requirements, checkers, error-correction circuits and/or isolation circuits are

connected in the communication paths between modules. Information transfers between modules is delayed by the need to complete error detection before transmission. This results in lower system performance due to increased clock cycle time or additional pipeline stages.

This section deals with the idea of transient fault recovery in a VLSI RISC processor where error checking is performed in parallel with the transmission of information between modules [YuT88]. The receiving module does not wait for checks to be completed. It proceeds with execution as the check is being carried out and the checking result is sent a few cycles later. The advantage of having checking in parallel is that checking delay is avoided.

But performing error checking in parallel with inter-module communication introduces a new problem. The state of the system starting with the receiving module may be polluted with damaged information before an error signal arrives. Therefore, it is necessary to back up processing to the state that existed just before the error occurred. This process of backing up the system several cycles in response to a delayed error signal is called *micro rollback* (shown in Figure 4.1). In order to support micro rollback, each module in the system buffers all the information necessary to undo the state changes that have occurred in the last few cycles.

Micro rollback differs from single instruction retry in that it occurs at a lower level on the basis of clock cycles rather than instructions. Since the system undoes cycles rather than instructions, it can be done at the logic level without keeping track of microprogram-level instructions and instruction pipeline conditions. Consequently the micro rollback

capability can be independently implemented in each module of the synchronous system following very simple specifications.

4.2.1 SUPPORT FOR MICRO ROLLBACK IN A REGISTER FILE

Under normal operating conditions a *write* may be performed in every cycle. It is impractical to preserve the state of the register file for N cycles by replicating the whole register file N times. An alternative method is proposed which minimizes the extra hardware while allowing the rollback of up to N cycles to be executed in one cycle.

Whenever the processor writes data to one of the registers, the address of the destination register and its new contents are stored in an N-word FIFO buffer, which is called a delayed write buffer – DWB shown in Figure 4.2. The register address is held in a content addressable memory (CAM) and the memory cells holding both the data and address can be shifted. During the register read phase of each instruction, the register addresses of the two operands are compared with the addresses of the registers stored in DWB. If there is a match and the valid bit in the CAM is set, the data of the matching register is gated to the corresponding internal data bus.

The DWB delays each write by N cycles before it is finally written into the register file. During every cycle, a new entry is made in the right most cell of the DWB. If a write occurs, the data is entered and the DWB position is marked as *valid*. If no write occurs, the DWB word is reserved and but marked as *invalid*. During every cycle, the oldest (leftmost) entry is written to its corresponding address in the register file if its valid bit is set, and discarded otherwise. In order to rollback p cycles ($1 \leq p \leq N$), the last p entries in the DWB are invalidated by clearing the valid bits.

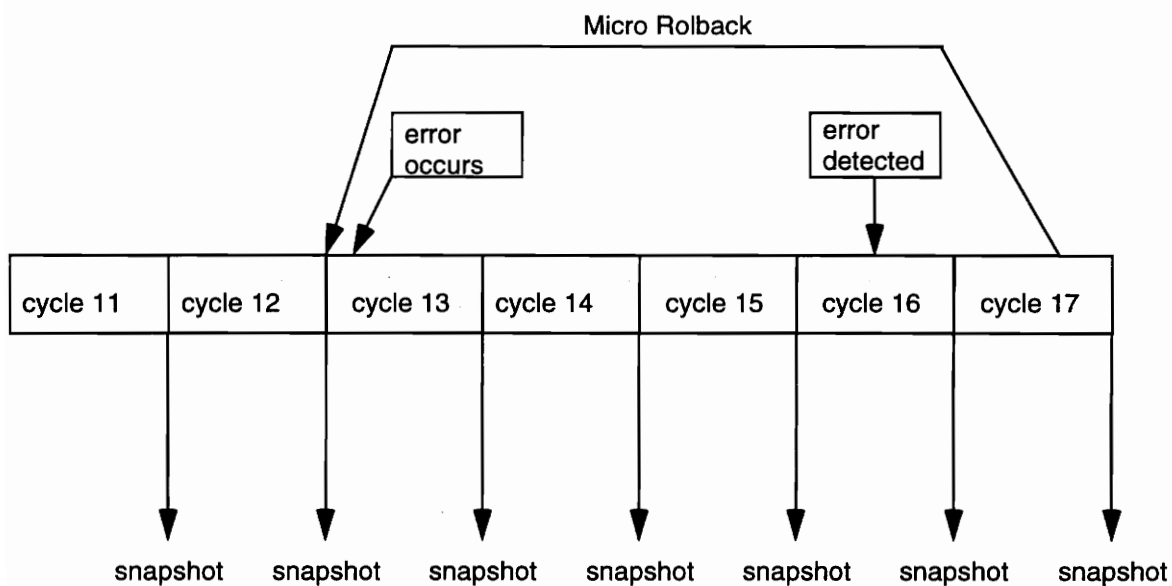


Figure 4.1 Micro Rollback of a Module

4.2.2 MICRO ROLLBACK OF INDIVIDUAL STATE REGISTERS

The VLSI processor has many other registers that contain part of the processor state. These include the program counter, instruction register, processor status word, pipeline latches etc. Since these registers are modified every cycle, support for micro rollback of up to N cycles requires each register to have N "backup registers" for saving the state up to the last N cycles. There are two methods for rolling back individual state registers:

- i. FIFO method
- ii. RAM method

In the FIFO method, as shown in Figure 4.3, the state register labeled as *current* is backed up by a set of N registers which are organized as a FIFO. The FIFO registers are transparent to the processor in the sense that all the *reads* and *writes* are performed are performed using the current register. At the end of each cycle, the contents of the current register is copied into the FIFO which pushes down all the state registers. In order to rollback p cycles, the first p registers of the FIFO are invalidated, then the first register marked as "valid" in the FIFO, identified through a priority circuit, is copied into the current register.

The RAM method uses a pointer to "next available backup register" in a small RAM to create a circular buffer as shown in Figure 4.4. The contents of the current register is saved in the RAM every cycle and the pointer is incremented. In order to rollback p cycles, p is subtracted from the current value of the pointer (modulus the number of elements in the RAM) and the result is used as an address of the register in the

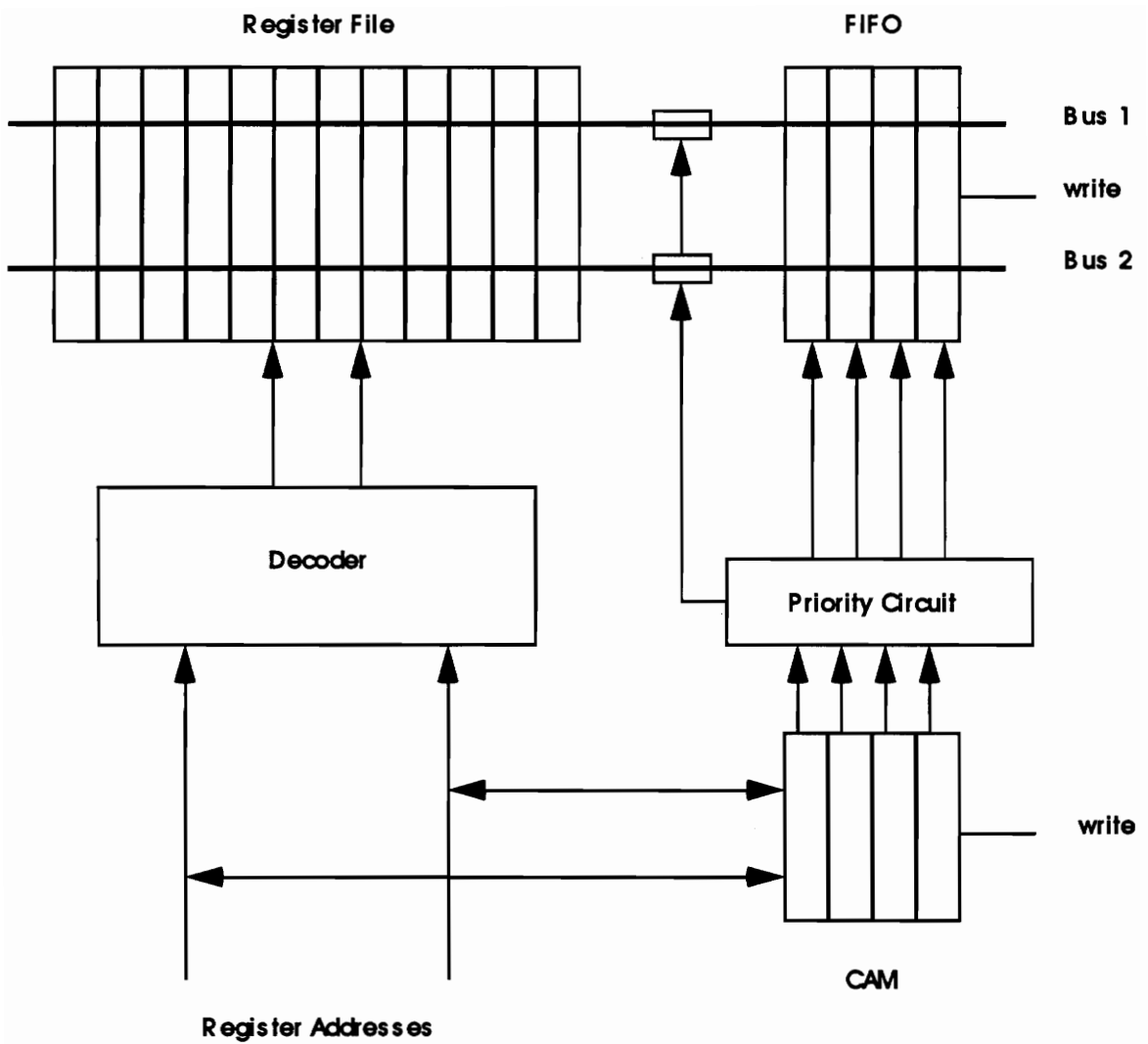


Figure 4.2 Register File with Support for Micro Rollback

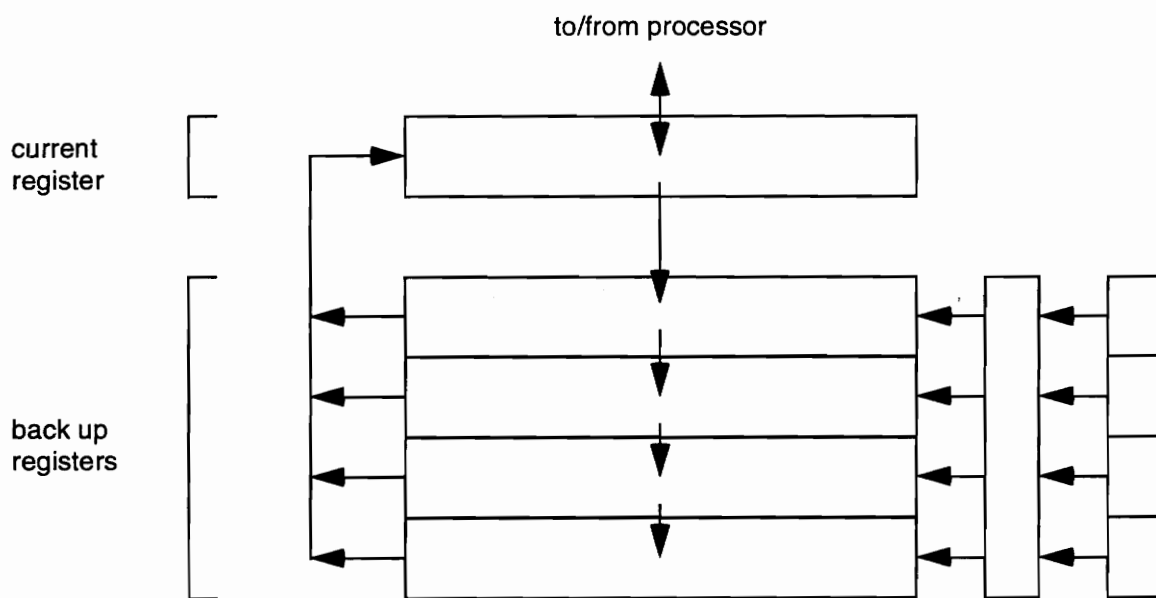


Figure 4.3 FIFO Method for Micro Rollback of Individual State Registers

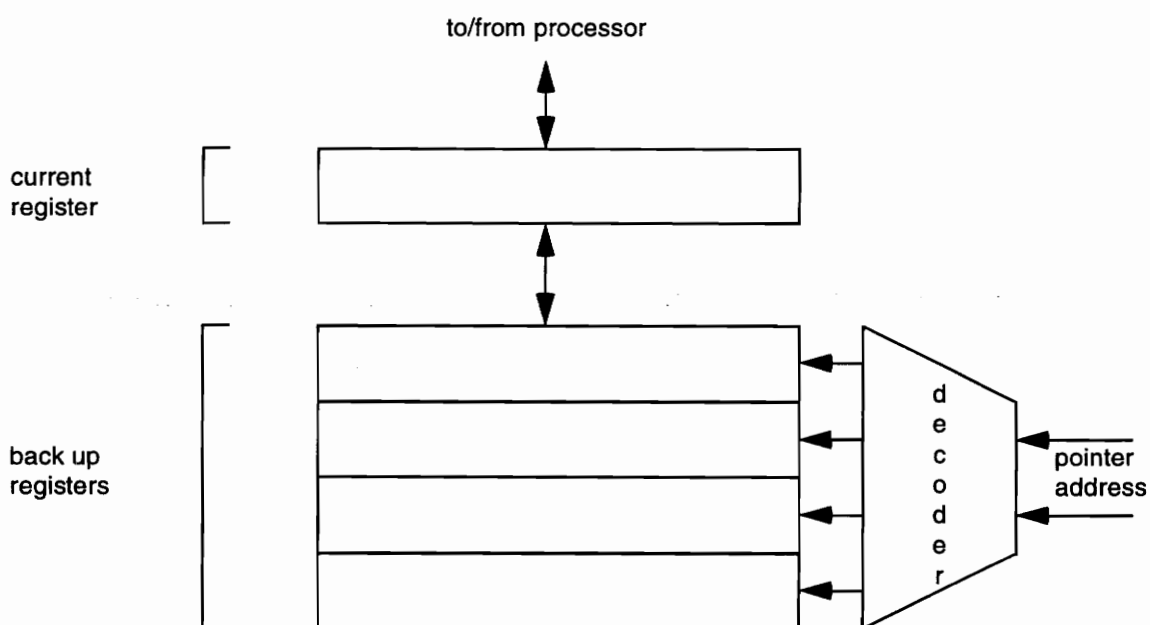


Figure 4.4 RAM Method for Micro Rollback of Individual State Registers

RAM, the contents of which are copied into the current register. The input to the address decoder of the RAM is same throughout the chip allowing a single centralized implementation of the counter and subtraction logic.

4.3 CONCLUSIONS

This chapter introduced the concepts of data rollback and checkpointing operations. Both the methods discussed in this chapter dealt with the idea of data rollback in a uniprocessor system. The first method called CARER is useful in cache based systems. This scheme takes advantage of the already available cache to implement rollback recovery. This scheme requires only a single checkpoint to be created for rollback operation. The checkpointing operation is not periodic in the sense that it is based on a timer.

In the second method, instead of rolling back instructions, cycles are rolled back. Hence, the scheme is called *micro rollback*. Error checking is done in parallel instead of serial. Since, rollback can be anywhere from one cycle to a maximum of N cycles, a checkpoint has to be created at the end of every cycle. Hardware overhead is high in this scheme because of duplication of individual registers N times. The control circuitry for implementation of rollback is also quite complicated.

Chapter 5 RECOVERY FROM TRANSIENT FAULTS IN WAVEFRONT PROCESSOR ARRAYS

A wavefront processor array is characterized by data flow across the array that resembles wave propagation. Each processing element in the array receives inputs from its neighbors on the north and west and acts as a secondary source of wavefronts for its neighbors in the east and south. The processing elements in the topmost row and the leftmost column receive their inputs from the input modules that are under the control of an external processor. The data transfer in the wavefront architecture is asynchronous in nature without any global clock synchronization.

Field studies have shown that transient faults occur far more frequently than permanent faults. In fact, transient faults account for 90% to 98% of all detected faults [SiSw82]. Recovering from transient faults in a processor array is important because over 90% of the time, a permanent fault reconfiguration algorithm will be removing a PE from the system that has only a temporary fault. Restarting the computation in the array from

the very beginning in response to the occurrence of a transient fault is not only inefficient but usually not possible for real time computations with limited input buffering.

5.1 FAULT DETECTION ASSUMPTIONS

We will consider two methods to detect transient faults in a processing element – totally self-checking systems and the time redundancy. In the time redundancy technique, data transmission is done without waiting for error detection to be complete. Time redundancy error detection is done in two steps. In the first step, normal computation is done and the result is passed to the neighboring processing elements. The second step involves recomputation with shifted operands and reverse shifting the obtained result. The results of both the steps are compared to find if an error has occurred. When a transient fault occurs in a processing element implementing this scheme, erroneous data has propagated to the neighbors before detection is complete. Therefore, this scheme requires a more complicated procedure for recovering from a transient faults and hence, is not preferred.

In a distributed computing environment such as a wavefront processor array, if a totally self checking scheme is implemented data contamination is avoided. If a transient fault occurs in a totally self checking circuit, then for any codeword input, the circuit either produces a correct codeword output or it produces a non-codeword. This ensures that the first erroneous data produced by the faulty processor will be detected. As it will be shown, an advantage of detecting an error as soon as it occurs is that rollback has to be initiated only in the faulty processor. Other processors must simply pause until the rollback is complete.

5.2 ROLLBACK ALGORITHM

There are two aspects of transient fault recovery in wavefront processor arrays:

1. Statesaving operation
2. Rollback operation

The state-saving is distributed in nature i.e., there is no centralized controller coordinating the activities of all the processing elements. Therefore, each processor must decide on its own when to save the processor state. The *statesave period* is the number of data inputs received after which a processor saves its state. If the statesave period is, for example eight, then any processor saves its state after every eight inputs. The statesaving operation itself depends on the complexity of the processing element. If the processing element consists of a simple ALU with a few registers then statesaving consists of copying the register contents into the memory. If the processing element includes memory with cache implementation, then the statesaving operation is more complex as in CARER [HuM87]. The amount of time required to save the state of the processing element is called the *statesave time*.

The rollback algorithm is initiated whenever a transient fault is detected by either the processing element or its neighbors. A transient fault is detected as soon as a non-codeword is produced by a processing element. The rollback algorithm is as follows:

1. As soon as a processing element realizes that it is in a faulty state, it stops all computation. It informs its immediate neighbors on the north, south, east and west about the occurrence of fault. This is shown in Figure 5.1.

2. The faulty processor rolls back to the last checkpoint and restarts its computation. For rollback to execute correctly, the faulty processing element not only has to save the state variables of the last checkpoint but it must also buffer all of the inputs it receives subsequent to the last checkpoint. When the processing element is about to create the $(n+1)^{\text{th}}$ checkpoint it still needs the inputs corresponding to the n^{th} checkpoint. This is because if the inputs corresponding to the $(n+1)^{\text{th}}$ checkpoint are non-codewords, it loses the n^{th} checkpoint inputs. The n^{th} checkpoint inputs are needed to start the rollback operation. Therefore, the minimum buffer size needed is equal to $(\text{statesave period} + 1)$. Any processing element needs to save only one checkpoint for rollback, so when the processor is about to save its $(n+1)^{\text{th}}$ checkpoint it still needs the inputs corresponding to the n^{th} checkpoint. The maximum buffer size needed is determined by the input data rate which will be discussed later.

3. Neighboring processing elements, upon receiving the error information, pass it on to their neighbors. The "stop propagation" information is propagated throughout the array to preserve the wavefronts while the faulty processor is undergoing rollback. The transmission of "stop propagation" information implies setting of appropriate stop bit in its neighbors. Each processing element of the array has associated with it four "stop propagation" bits – east, west, north and south, corresponding to the direction from which "stop propagation" information is received and also a bit called "stop operation." The "stop operation" bit is also set when one of the "stop propagation" bits is set. The utility of the "stop operation" bit is explained later. After the processing element receives the "stop propagation" information, it checks to see which of the four stop bits has been set. The stop bits are polled only when the processing element has finished the current execution

task. If the west bit of any processor is set then it informs its east neighbor of "stop propagation" by setting the neighbor's west bit. Similarly, if the east bit of any processor is set, it informs its neighbor on the west by setting the neighbor's east bit. The transmission of "stop propagation" in east and west directions is shown in Figure 5.2. When any of the south or north bits of a processing element are set, then the "stop propagation" information is transmitted in a different manner. This is to ensure that "stop propagation" is transmitted in all four directions. If the south bit of any processor is set, it informs its neighbors on the east, west and north by setting their west, east and south bits respectively. If the north bit of any processor is set, then it informs its neighbors in the east, west and south. The transmission of "stop propagation" for this case is shown in Figure 5.3. The combination of the above two cases is shown in Figure 5.4. If any processing element in the topmost row or leftmost column receives "stop propagation" information, then it informs the I/O controller to stop pumping data into the array.

4. Any processing element receiving the "stop propagation" information, stops communication of data until the faulty processing element has recovered. This is done to preserve the wavefronts propagating across the array. If the processing elements in the array are allowed to communicate during a rollback operation, then the faulty processing element undergoing rollback may receive more inputs than it can buffer, thus losing some critical data.

5. When the "stop propagation" information reaches the boundary of the processor array, the I/O controller is informed to temporarily suspend the input operation. In real-time applications, the data that cannot be sent to the array has to be buffered by the I/O controller.

F: Faulty Processor

1: Processors informed in first iteration

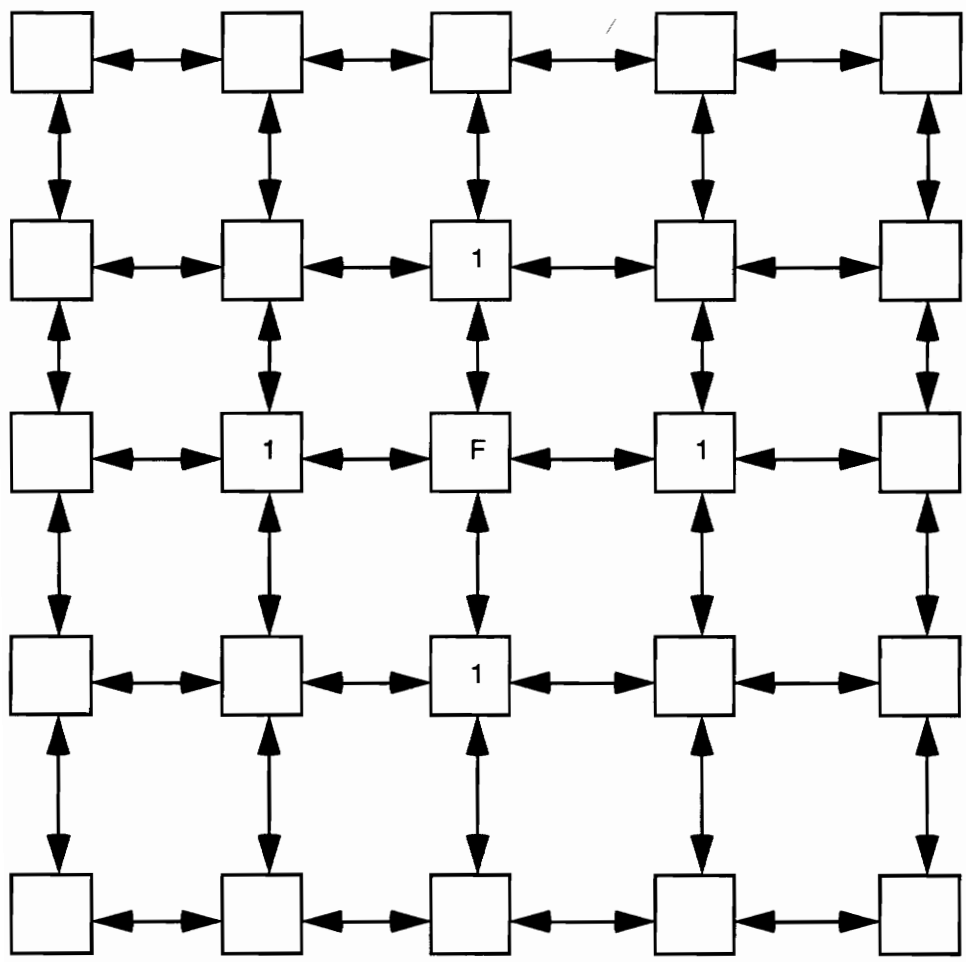


Figure 5.1 Transmission of "stop propagation" by Faulty Processor

F: Faulty Processor

1: Processors informed in first iteration

2: Processors informed in second iteration

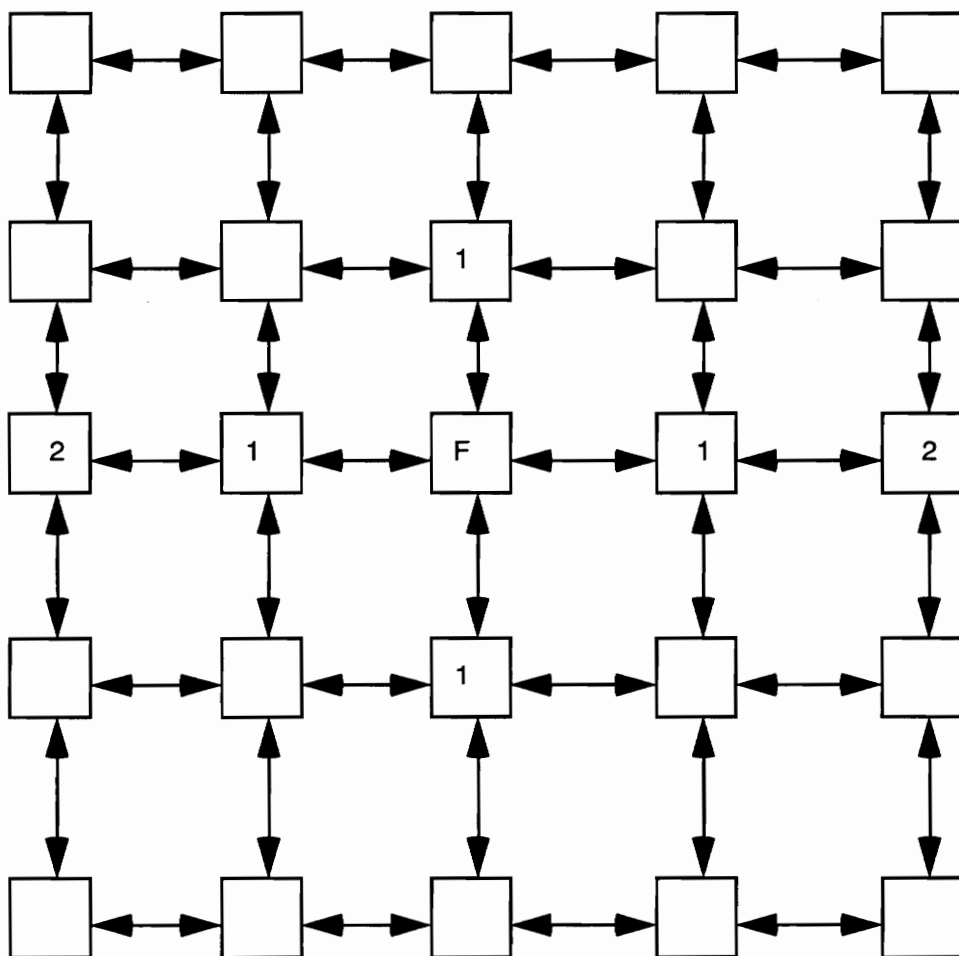


Figure 5.2 Transmission of "stop propagation" for East and West Neighbors

F: Faulty Processor

1: Processors informed in first iteration

2: Processors informed in second iteration

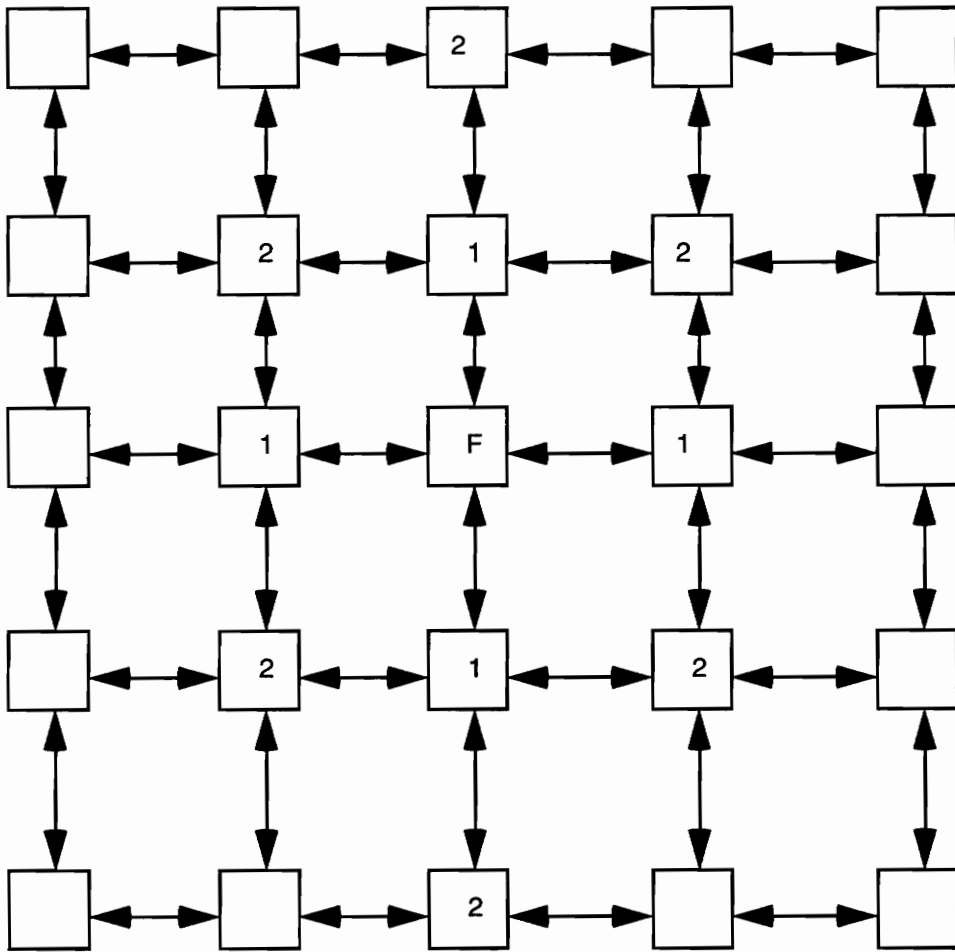


Figure 5.3 Transmission of "stop propagation" for North and South Neighbors

F: Faulty Processor

1: Processors informed in first iteration

2: Processors informed in second iteration

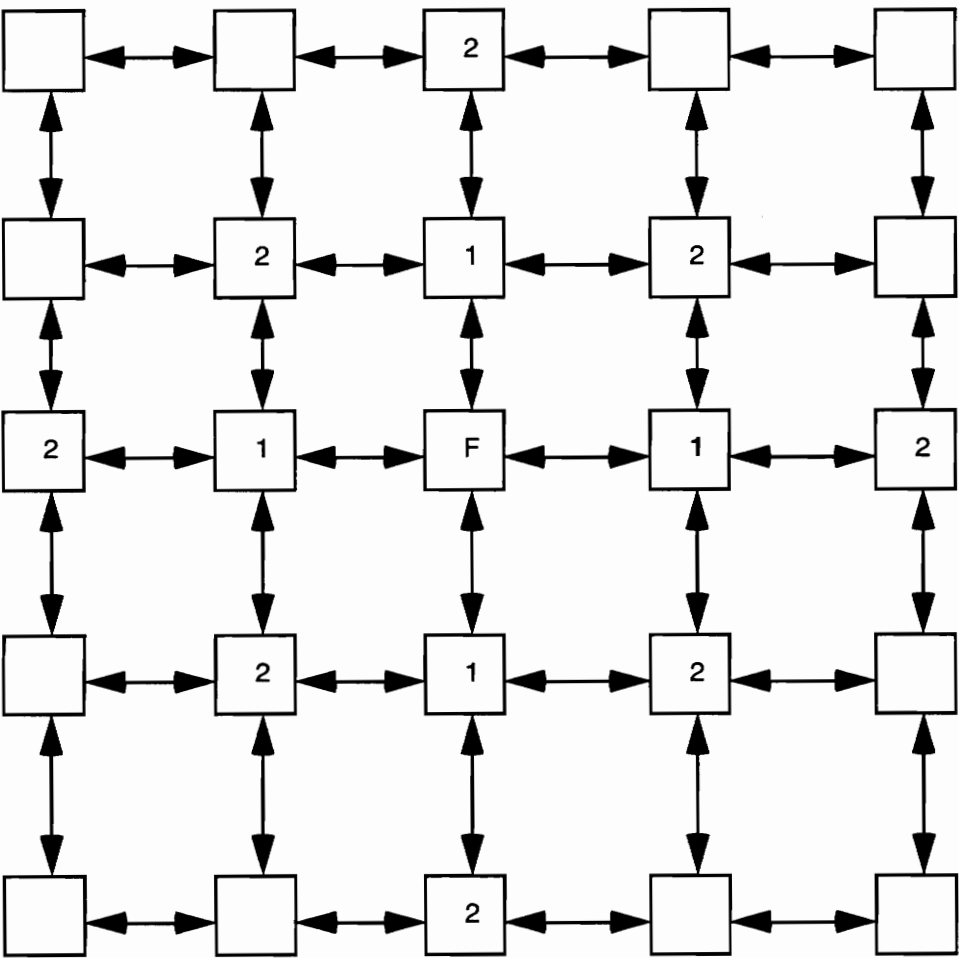


Figure 5.4 Transmission of "stop propagation" through the Array

6. After the faulty processing element has successfully completed the rollback operation, it clears the "stop operation" bit of all its four neighbors. Any processing element after receiving the "stop propagation" information continues to poll the "stop operation" bit. When the "stop operation" bit is cleared, it polls the stop bits to find which one of them is set. If the east stop bit is set, then the processing element clears "stop operation" bit of its west neighbor and also clears its own east stop bit. After the "stop operation" bit is cleared if the processing element finds its west stop bit is set, then it clears the "stop operation" bit of its east neighbor. If the south stop bit is set after the "stop operation" bit is cleared, then the processor clears the "stop operation" bit of its north, east and west neighbors. When the north stop bit is set after the "stop operation" bit is cleared, the processing element clears the "stop operation" bit of its neighbors in the south, east and west. Whenever the processing element clears the "stop operation" bit of its neighbor/neighbors, it clears its own stop bit. The "direction" stop bit is not cleared by the neighbor because the processing element would not be able to select its neighbors for clearing the "no operation" bit. Hence, the "restart" signal is transmitted through out the array by clearing the "no operation" bit. The direction in which "stop operation" bit is cleared by the processing element depends on which of its stop bit is set and is similar to the way in which "stop propagation" information is transmitted. The "stop propagation" and the "restart" signal are transmitted in the way discussed, to ensure any processing element receives the stop and restart signals from only one direction. The processing elements at the boundary of the array finally convey the "restart" information to the I/O controller so that normal operation can continue again.

7. Since the processing elements are totally self checking, the first erroneous data coming out of the faulty processor is a non-codeword. This is recognized by the

neighboring processing elements when they receive erroneous data. This ensures that processing elements do not initiate rollback because of a faulty neighbor. Therefore, there is no rollback propagation.

8. If a processing element continues to produce non-codewords after rollback then it is diagnosed as having a permanent fault. At this stage, a permanent fault reconfiguration algorithm is invoked and the faulty processing element is removed from the logical array. After permanent fault reconfiguration, rollback cannot be implemented because the information associated with the faulty processing element is lost.

5.3 HARDWARE OVERHEAD

The processing element configuration modified from Figure 2.4 for statesaving operation is shown in Figure 5.5. For recovering from transient faults, the first step is to design all the processing elements of the array as totally self-checking systems. The details of designing a totally self-checking functional circuit can be found in [SmL83].

When the state of the processor needs to be saved during the checkpointing operation, extra memory is needed to save the contents of registers such as the program counter, the accumulator etc. Since only one checkpoint is to be saved at any instant of time, the extra memory needed is equal to the number of registers to be saved. The statesave memory shown in Figure 5.5, corresponds to the extra memory needed for statesave operation. If the processing elements of the array include cache structures then a technique such as CARER [HuM87] can be used.

1 Control Bus: 2 bi-directional signals – direction stop bit and stop operation bit

1 Data Bus: 1 signal – 1 serial line

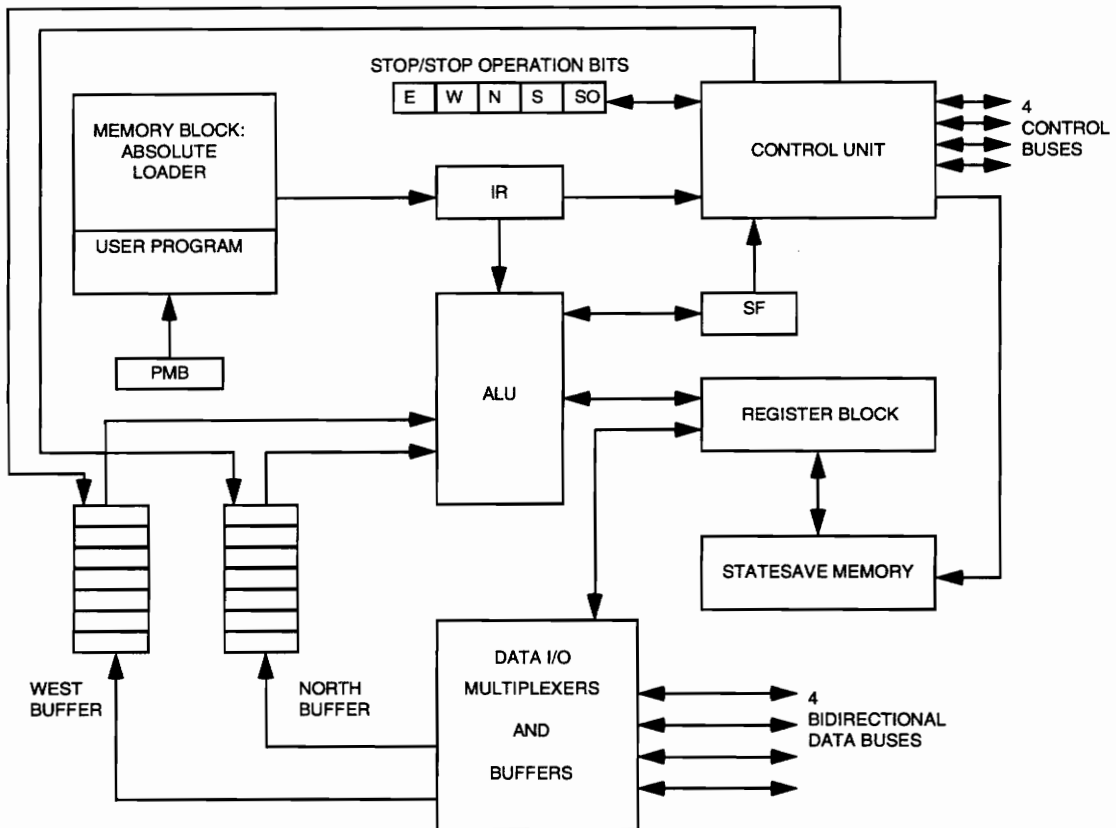


Figure 5.5 Processing Element with Hardware Overhead

After the state of the processor has been saved, all the inputs received by the processor between the present checkpoint and the next need to be buffered. If a transient fault occurs and the processor rolls back to the checkpoint, it must repeat all calculations starting at the checkpoint. Therefore, the inputs during that time are needed. Two buffers are needed – a north buffer for inputs from the north and a west buffer for inputs from the west. Both buffers are implemented as a shift register and are shown in Figure 5.5. A lower bound on the size of each buffer is determined by the state save period. The minimum size of the buffers is equal to the (statesave period + 1). An upper bound on the size of each buffer was experimentally determined and depends on the location of the processing element in the array, the input period and the execution time. The simulation results for the buffer size will be discussed in Chapter 6. Two bits are associated with each register of the buffer. The first bit is set when the processing element has used the corresponding input for *computation* while the second bit is set to identify it as a *checkpoint*.

After a transient fault has been detected in a processing element, "stop propagation" is transmitted throughout the array. To identify the direction from which the "stop propagation" is received, four stop bits are needed – north (N), south (S), east (E) and west (W). The manner in which these bits are set and cleared is explained in the *rollback algorithm*. Other than the four stop bits mentioned above, another bit called "stop operation" (SO) is needed in each processor for transmitting the "restart" signal after rollback is over. Hence, between any two processors two bi-directional control signals are needed – "direction" and "no operation" signals (shown in Figure 5.5). When a processing element wants to transmit "stop propagation" to its neighbor, it sends a pulse on both control lines. One of the direction stop bits – E, W, N or S and the "stop operation" bit

SO are both set in the neighboring processing element. When a processing element wants to transmit the "restart" signal to its neighbor, it sends a pulse only on the "stop operation" control signal line which clears the "stop operation" bit – SO in the neighbor. After the "stop operation" bit SO is cleared, the neighbor polls its own direction stop bits to find which one of them is set. Depending on the direction stop bit set, it clears the SO bit of its neighbor or neighbors and clears its own direction stop bit.

In a processing element with no fault-tolerance for transient fault recovery, handshaking is needed for data transfer. The details of data transfer operation are explained in Section 2.5. In the processing element modified for transient fault recovery, no handshaking signals required between processors for data communication because the data is directly transferred to a shift register (north buffer or west buffer) and hence never overwritten.

5.4 SIMULATION PROGRAM

The simulation program for the wavefront processor array is written in C. The wavefront array is a two dimensional network of processing elements connected in a mesh configuration. An input module is connected to each processing element in the topmost row and leftmost column of the array. Each input module feeds data to its processing element at regular intervals of time. The time interval between two successive inputs is known as the *input period*. After the processing element has received its inputs from the north and west direction, it waits for a period of time called *execution time* before generating outputs for the west and south direction. After every statesave period, each

processing element in the array creates a checkpoint. The amount of time required for the statesave operation is the *statesave time*.

All the events in the wavefront processor array are scheduled by using a "timing wheel." The timing wheel is a one-dimensional array with a linked list at each indexed location. The linked list consists of all events scheduled at that time.

$$\text{index of the array} = (\text{current time}) \bmod (\text{wheel size})$$

where wheel size is the size of the timing wheel array. Wheel size is determined by the task that takes the most time in the wavefront processor array. In fact,

$$\text{wheel size} = \text{time taken by the longest task} + 1.$$

5.5 CONCLUSIONS

In this chapter, the rollback algorithm and the issues pertinent to it have been stated. An important criterion in the rollback algorithm is that the processing elements are designed to be totally self-checking. In a totally self-checking circuit erroneous information is detected immediately. This ensures that there is no rollback propagation and rollback is initiated only in the faulty processor. The rollback algorithm also maintains continuity of computation after the occurrence of a transient fault so the algorithm is useful for real-time applications. By initiating transient fault recovery before permanent fault reconfiguration, the possibility of removing a processing element having only a transient fault from the logical array is eliminated. The reliability of the array is enhanced, which is the entire motive of the reconfiguration algorithm.

Chapter 6 RESULTS

In this chapter, the results of the simulation runs are discussed. The inputs to the simulation program are input period, execution time, statesave time, statesave period, fault location and time at which the transient fault is to occur. For simulation runs, the execution time in each processing element was chosen by finding the time taken in a typical processor for floating point multiplication and addition. This is because most signal processing applications involve floating point multiplication and addition. In the i-486 processor, it takes twenty clock cycles for the above two operations to be executed. So the execution time in each processing element of the array is chosen to be twenty time units.

During the simulation runs, few errors were noticed. Fixing one of the errors involved modifying the rollback algorithm. Before the error was discovered, the transmission of "stop propagation" and "restart" signal by any processing element was done by informing the processing elements in all four directions – north, south, east and

west. Hence, any processing element received "stop propagation" information from more than one direction. Only one stop bit was needed for this purpose. During one of the simulation runs it was noticed that a processing element received "stop propagation" information and after rollback was over it received the "restart" signal. Then again it received the "stop propagation" signal which had propagated through a different path. This caused the affected processing element to stop and the wavefront processor array lost its normal data flow synchronization. This problem was solved by ensuring that both the "stop propagation" and the "restart" signals are transmitted through unique paths.

In a wavefront processor array with no fault-tolerance for transient fault recovery, if the execution time in each processing element is twenty time units, an input period of twenty time units maintains constant throughput. Only one set of inputs needs to be buffered. But, when the statesaving operation is introduced for transient fault recovery, the statesave time introduces a time overhead after every statesave period number of inputs have been received by a processing element. Hence, for an execution time of twenty time units, an input period of greater than twenty time units is needed.

6.1 SKEW AFFECT

The *skew affect* is the lag between two inputs coming from the west and north direction for the processing elements along the topmost row and leftmost column. These processing elements receive one of their inputs from I/O buffers. The processing elements in the topmost row receive their north inputs from I/O buffers while the processing elements in the leftmost column receive their west inputs from the I/O buffers. The lag is

introduced because of the statesave operation and increases as we move away from the processing element located at (0,0) along the topmost row and the leftmost column. This is because lag is introduced in each processor along the first row and column.

$$\text{Skew} = | \text{input number from west} - \text{input number from north} |$$

For example, at any processor, if at the topmost location of north buffer the tenth input has arrived while at the topmost location of west buffer only the sixth input has arrived, then the skew = $|6 - 10| = 4$.

The skew affect for different input periods is shown in Figure 6.1. The processing elements chosen for observation are (5,0), (9,0) and (14,0) – all along the leftmost column. When the input period is close to the execution time, skew is high. For example, consider the case in Figure 6.1 where, execution time = 20, input period = 22, statesave period = 6 and state save time = 4. The processing element located at (0,0) receives the first set of inputs at time $t = 0$. After a statesave operation to create the first checkpoint, it starts its first execution. The processor located at (1,0) receives its first input (north) from (0,0) at $t = 0 + \text{statesave time} + \text{execution time} = 0 + 4 + 20 = 24$. The processing element at (1,0) receives its first input from the west at $t = 0 + \text{input period} = 0 + 22 = 22$. Hence, the processing element located at (1,0) has a skew of 1 from $t = 22$ to $t = 24$. The skew increases as we move along the topmost row or leftmost column of the array. For the wavefront processor array to perform execution properly with statesave operations, it is important that skew be stable with time at any processor location. The buffer size has to be modified to accommodate the skew affect. The details are discussed in later sections.

X-axis: Input Period

Y-axis: Skew

Statesave Period: 6

Statesave time: 4

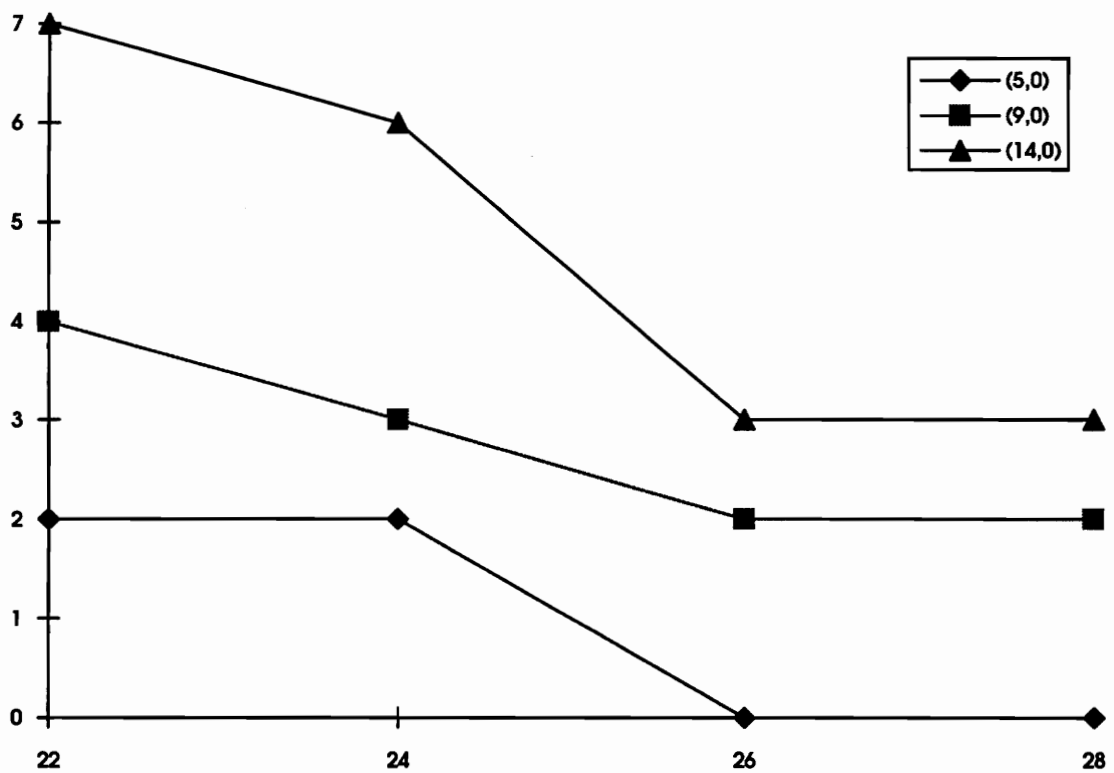


Figure 6.1 Variation of Skew with Input Period

The skew affect is not observed in processing elements away from the top row and leftmost column of the array. Any processor away from the top row and leftmost column of the array receives its inputs from its neighbors which save their state simultaneously. Hence, the neighbors delay their outputs by the same margin resulting in no skew in the receiving processing element.

From Figure 6.1, when the input period = 22, the skew observed at location (14,0) is seven. This implies that at the processing element located at (14,0), seven more inputs have arrived from the west direction than from the north direction. To retain inputs from both north and west direction for computations, the buffer size for the west should be greater than north by at least seven. As the input period is increased, the inputs from the north direction have more time to catch up and hence, the skew reduces. For example, when the input period = 28, the skew at location (14,0) is three. This skew reduction has come at the expense of increased input period which implies reduced throughput.

The skew increases as the location of the processing element moves away from (0,0) location along the topmost row and leftmost column. Therefore, as the size of the array increases, the buffer size needed in the processing element to accommodate the skew also should be increased.

6.2 VARIATION OF BUFFER SIZE WITH STATESAVE PERIOD

The buffer size in a processing element is a function of statesave period for any given input period. As shown in Section 5.2, the lower bound on buffer size is (statesave

period + 1). The variation of buffer size with state save period for different input periods is shown in Figure 6.2.

When the input period is close to the execution time and the state is saved frequently, then the buffer size needed increases with time. In Figure 6.2, for an input period of twenty two, the buffer size needed increases with time for statesave periods equal to one and two. The corresponding points are therefore not shown in Figure 6.2. The buffer size becomes stable for statesave period equal to four. When the input period is increased to twenty four, the buffer size becomes stable for statesave period equal to two. As evident from Figure 6.2, the buffer size at (14,0) processing element increases with increasing statesave period. The lower bound on buffer size (statesave period + 1) in a processing element increases linearly with statesave period. The difference in lower bound and the values seen in Figure 6.2 is explained in the next section.

The difference between lower bound and values seen in Figure 6.2 on the buffer size for a given statesave period is because of the skew affect. For example, when input period = 22 and statesave period = 4, the buffer size = 11. The skew affect increases the buffer size needed from five (lower bound) to eleven (upper bound). When the input period is increased to thirty, the buffer size needed drops to five (upper bound) which is equal to the lower bound value. At this input period no skew is observed. But, this decrease in buffer size is achieved at the expense of decreased throughput. In fact, buffer size and skew are related as

$$\text{buffer size} = \text{statesave period} + \text{skew} + 1.$$

X-axis: Statesave Period

Y-axis: Buffer Size

Execution Time: 20

Statesave Time: 4

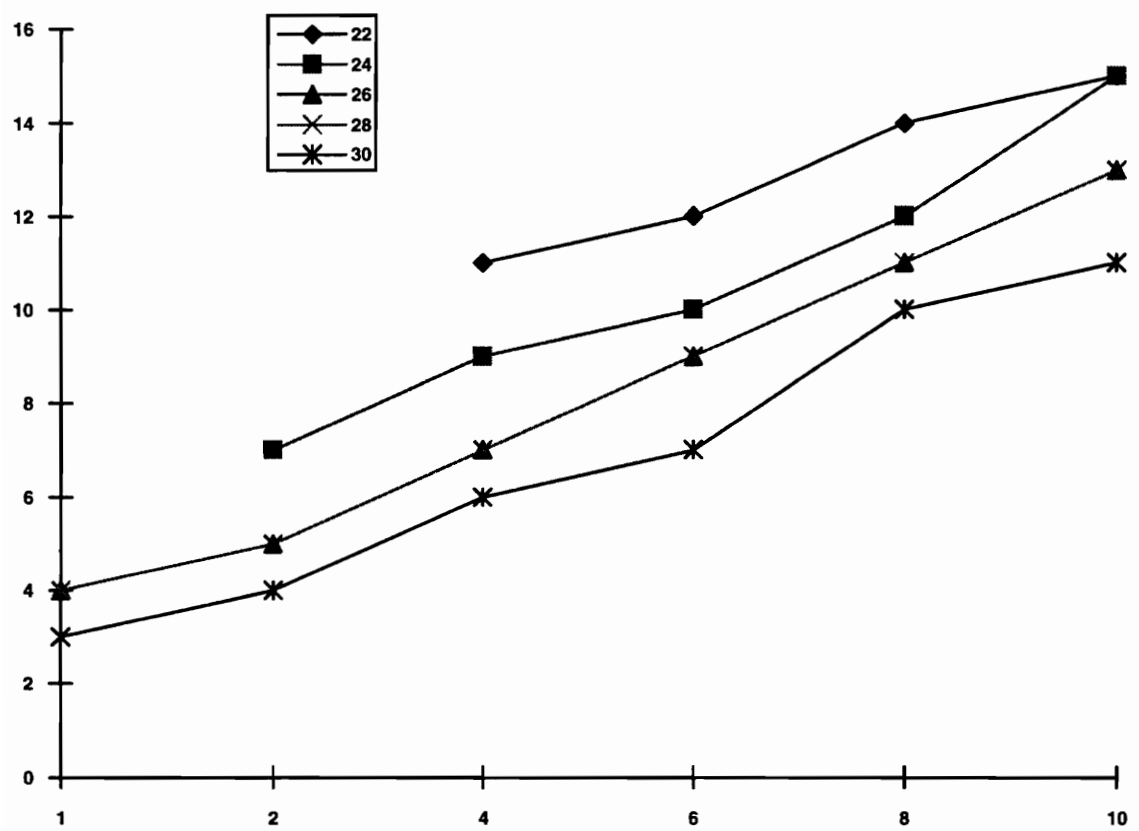


Figure 6.2 Variation of Buffer Size with Statesave Period

X axis: Statesave Period

Y-axis: Buffer Size

Execution Time: 20

Statesave Time: 8

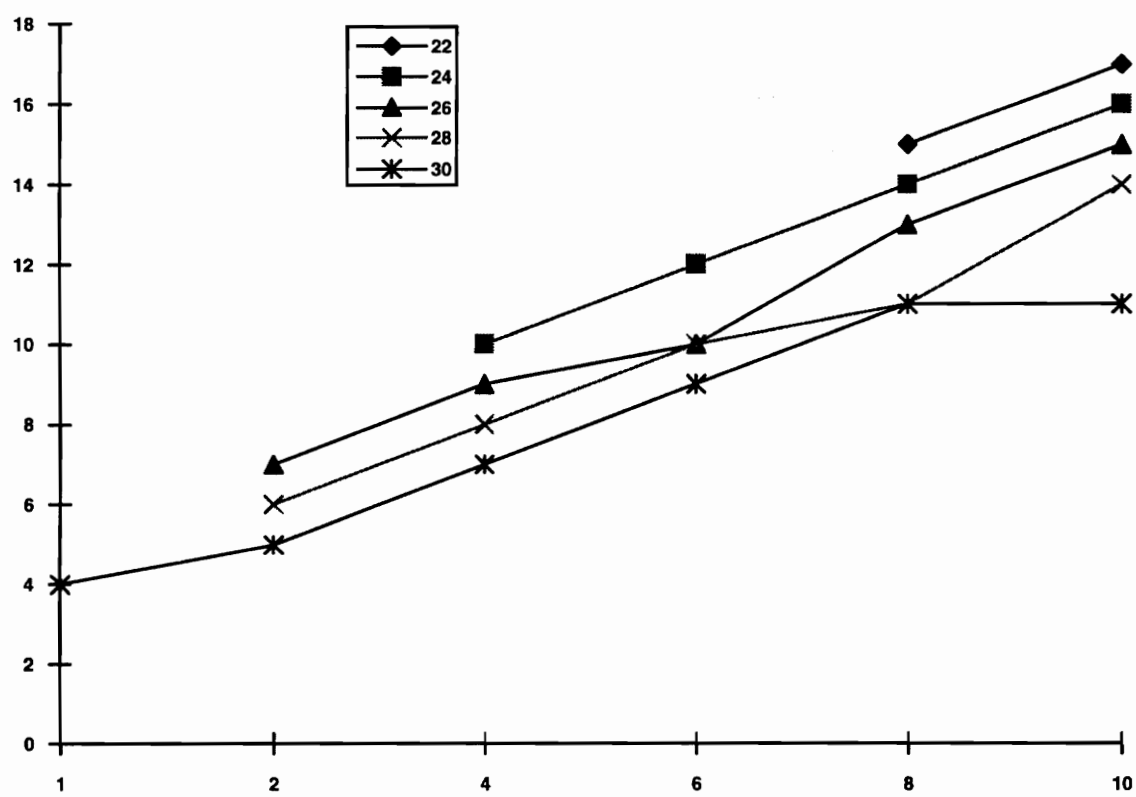


Figure 6.3 Variation of Buffer Size with Statesave Period

The variation of buffer size with statesave period when statesave time is increased from four to eight is shown in Figure 6.3. For the input period of twenty two, the buffer size becomes stable only for statesave period of eight and higher. The points corresponding to statesave periods less than eight are therefore not shown in Figure 6.3. In Figure 6.2, when statesave time is four and input period is twenty two, the skew becomes stable for lesser value of statesave period i.e., four. Hence the threshold value of statesave period for skew stability at any input period decreases with a decrease in statesave time.

On comparing Figure 6.2 and 6.3, for the same value of input period, we find that buffer size needed in Figure 6.3 is higher. When the statesave time is increased from four (Figure 6.2) to eight (Figure 6.3), time overhead increases, which introduces an extra skew. Buffer size in a processing element increases linearly with skew. Hence, the affect of increasing statesave time is to increase the buffer size in a processing element.

6.3 VARIATION OF SKEW WITH STATESAVE PERIOD

In Figure 6.3, for an input period of twenty two, the absence of points for statesave period less than eight indicates that the buffer size is unstable for statesave period less than eight. When the difference between input period and execution time is small and the state is saved frequently, the skew at any processor location in the top row or leftmost column increases with time. At any given input period, the threshold for skew stability is determined by the statesave period. After the threshold value of statesave period has been reached, if the statesave period is increased, then it has a very slow affect on the skew. This behavior is shown in Figure 6.4 for different input periods. When the

statesave period is increased from four to twenty eight for an input period of twenty four, the skew observed at location (14,0) remains constant. For an input period of thirty, the skew observed at location (14,0) remains constant at one when the statesave period is increased from four to twenty four. The skew finally drops to zero when the statesave period reaches a value of twenty eight. Hence, the decrease in skew because of increase in statesave period is very slow.

After inspecting Figure 6.4, it becomes obvious that for a given statesave period, skew is affected more by the increase in input period. For example, when the input period is twenty two and the statesave period is four, the skew observed at location (14,0) is eight. When the input period is increased to thirty at the same value of statesave period, the skew observed at location (14,0) is only one. Hence, the skew decreases rapidly with an increase in input period for the same value of statesave period. The difference between execution time and input period is large when the input period is thirty, so the statesave operation introduces very little lag in the throughput and skew is very small.

The surprising conclusion drawn from Figure 6.4 is that increasing the statesave period to reduce time overhead does not reduce the skew significantly. For example, for input period of twenty two and statesave period of eight, the skew is seven. When the statesave period is increased to twenty eight, the skew is still seven. Since there is no reduction of skew because of increased statesave period, but there is an increase in buffer size with statesave period, it is better to have a small value of statesave period with applications. In other words, it is better to save the state more often as long as the value of statesave period falls above the threshold value, as identified in the previous section.

X-axis: Statesave Period

Y-axis: Skew

Execution Time: 20

Statesave Time: 4

Location of Processor: (14,0)

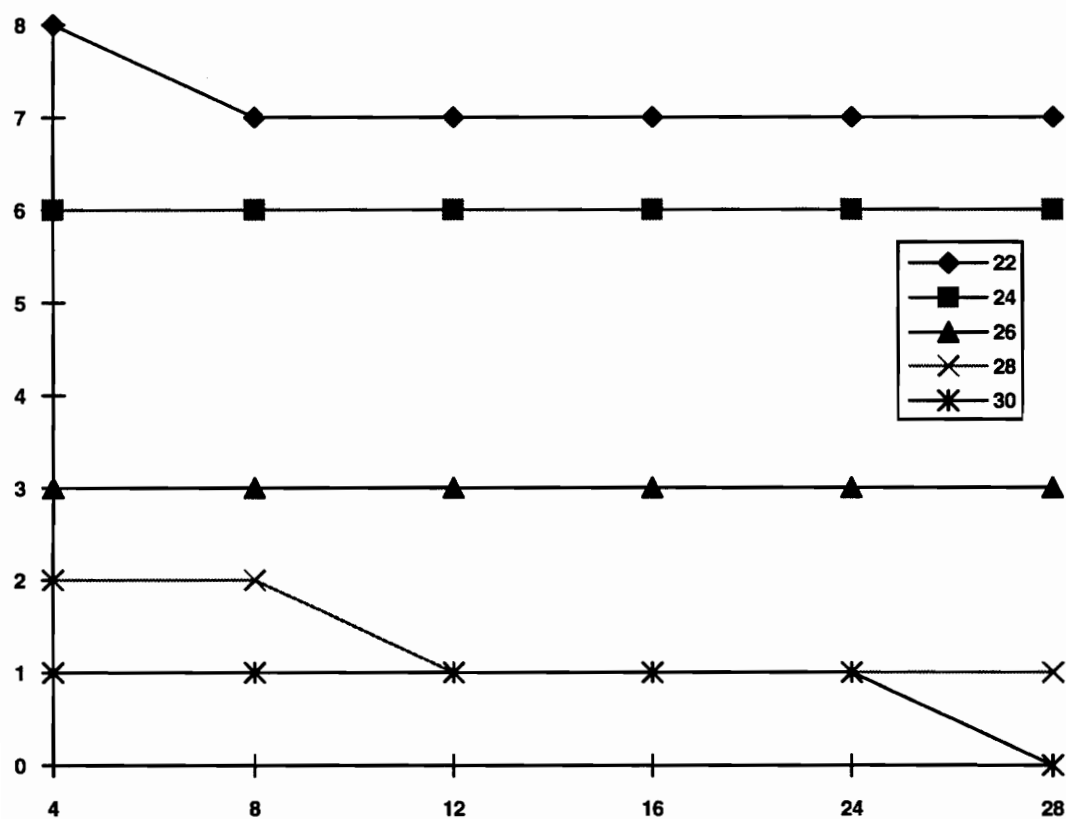


Figure 6.4 Variation of Skew with Statesave Period

Hence, the skew is affected more by the difference between the input period and execution time than by the statesave period.

6.4 SKEW AFFECT AFTER ROLLBACK

There were many simulation runs done to find the affect of rollback on the data-flow operations of the wavefront processor array. In the example chosen for explaining the skew affect after rollback, the statesave period is eight. In the chosen example, when a transient fault is introduced at processing element located at (4,0) at time $t = 1000$, the recovery initiated at the faulty processor (4,0) rollsback the inputs by eight which is equal to the statesave period. This is the maximum possible rollback the affects observed are worst case. The other specifications for the simulation run are:

Input Period: 28

Execution Time: 20

Statesave Time: 8

Statesave Period: 8

The skew affect before and after rollback is shown in Figure 6.5. Before rollback, the skew increases as the location of a processing element is moved farther away from (0,0) along the leftmost column. The skew observed at (2,0) is zero while the skew at (14,0) is four. This affect has already been explained in Section 6.1. After rollback, there is still no skew observed at the processing element located at (2,0). The skew at the faulty processing element (4,0) after rollback also remains unchanged. But, for the processing elements following (4,0) along the leftmost column, the skew after rollback is more than the skew before rollback. For the processing element located

X-axis: Location of Processing Element

Y-axis: Skew

Input Period: 28

Execution Time: 20

Statesave Time: 8

Statesave Period: 8

Fault Location: (4,0)

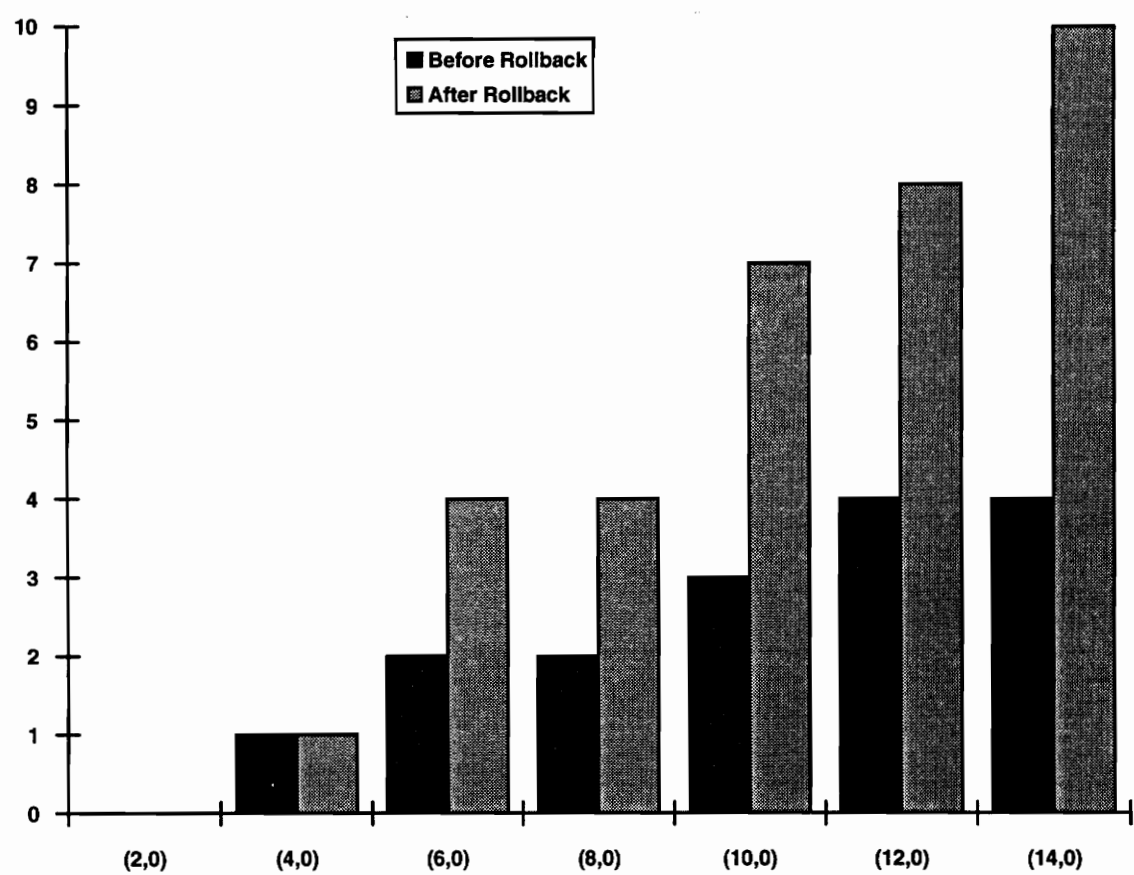


Figure 6.5 Skew Affect Before and After Rollback

at (14,0), skew increases from four to ten after rollback has occurred.

After rollback is successfully completed at the processing element located at (4,0), it sends the "restart" signal to its neighbors, which is again transmitted by the neighbors to their neighbors. The "restart" signal takes only one clock cycle to propagate across one processor. When the "restart" signal reaches the I/O buffers, the buffers start transmitting data to their respective processing elements. The processing element located at (8,0) receives the input from the west (I/O buffer) one clock cycle after it has transmitted the "restart" signal to the I/O buffer. The "restart" signal takes four clock cycles to propagate from (4,0) to (8,0). It takes at least $(3 * \text{Execution Time in a Processing Element})$ after rollback has occurred at (4,0) for the processing element located at (8,0) to receive the first input from its north neighbor (7,0). The difference in time between the arrival of inputs from the west (I/O buffer) and north (7,0) introduces extra skew after rollback.

As explained in the previous paragraph, after rollback has occurred, the skew increases as the location of a processing element moves away from location (0,0). The buffer size needed at location (14,0) before rollback = statesave period + skew + 1 = 8 + 4 + 1 = 13. After rollback has occurred, buffer size = statesave period + skew + 1 = 8 + 10 + 1 = 19. Hence after rollback, the buffer size needed in a processing element is greater. Hence, the size of buffer needed at any processing element location is determined by the skew after the rollback operation.

6.5 CONCLUSIONS

In this chapter, the different affects of the statesaving and rollback operations are discussed. The statesaving operation introduces a time overhead in the normal execution task of the processing element. Input period has to be increased to accommodate the statesaving operation which results in reduced throughput. Skew is an important affect resulting from the statesaving operation. Skew results in increased hardware overhead because of the increased buffer size needed. Skew can be reduced by increasing the input period which implies reduced throughput. The rollback operation also introduces skew, which should be taken into account while deciding the buffer size in any processing element of the array.

Chapter 7 CONCLUSIONS

The aim of the thesis is to develop with a scheme to recover from transient faults in wavefront processor arrays. Wavefront processor array is a systolic array without any global clock synchronization. Since there is no global clock distribution in a wavefront processor array, it is truly scalable.

The key to transient fault recovery is the error detection scheme. Two schemes for detection of transient faults are discussed in the thesis – totally self-checking scheme and time redundancy approach. The advantage of a totally self-checking circuit is that erroneous data is detected as soon as it is output by a processing element. Transient fault recovery techniques, using rollback, in uniprocessor systems are discussed to give insight into the statesaving operation. Two methods are presented – error recovery by rollback in cache based systems and rollback recovery in a VLSI RISC processor.

Chapter 5 details the rollback algorithm in a wavefront processor array. The algorithm discussed has an advantage that rollback is initiated only in the faulty processing element. Hence, the control circuitry needed to implement the rollback recovery is simple compared to when rollback has to be initiated in more than one processor. The rollback operation is useful for real-time applications. The rollback algorithm introduces time overhead because of the statesaving operation. To accommodate the time overhead throughput has to be reduced. The hardware overhead associated with the rollback operation consists of – statesave memory, west buffer, north buffer and the control bits. The control signals needed are the direction stop bits and the "no operation" bit.

The rollback algorithm was tested by writing a simulator program in C. Simulation runs were done to test – the skew introduced by the statesaving operation, variation of buffer size with statesave period, variation of skew with statesave period and skew affect after rollback. The statesaving operation has a direct affect on both the throughput and the hardware overhead. If an application needs a very high throughput then the input period should be low. To achieve minimum hardware overhead at the highest possible throughput, statesave operation should be done frequently. The lowest value of statesave period is the threshold of skew stability. When the input period is low, skew is high which implies buffer size in a processing element is high. To achieve a low value of hardware overhead because of buffer size, input period has to be correspondingly increased. Hence, a low value of input period results in high throughput but large hardware overhead in terms of buffer size and vice versa when input period is high. After the threshold value of skew stability has been reached, increasing the statesave period has very little affect on skew. The maximum size of buffer needed for the processing element is decided by the skew after rollback has occurred.

Before the rollback algorithm can be implemented in a wavefront processor array, the processing elements need to be made totally self-checking. Work needs to be done in the design of totally self-checking circuits. If the design of totally self-checking circuits is very complex, then error detection by using time redundancy can be adopted. The rollback algorithm will then have to be modified so that rollback recovery can be initiated in more than one processor.

BIBLIOGRAPHY

- [AnM73] D. A. Anderson and G. Metze, "Design of totally self-checking circuits for m-out-of-n codes," *IEEE Transactions on Computers*, Vol. C-22, pp. 263-269, Mar. 1973.
- [Av71] A. Avizienis et. al., "The STAR computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE Transactions on Computers*, Vol. C-20, pp 1312-1322, Nov. 1971.
- [BeB92] K. P. Belkhale and P. Banerjee, "Reconfiguration Strategies for VLSI Processor Arrays and Trees using a Modified Diogenes Approach," *IEEE Transactions on Computers*, Vol 41, No. 1, pp 83-92, Jan. 1992.
- [ChF86] M. Chean and J. A. B. Fortes, "FUSS: A Reconfiguration Scheme for Fault Tolerant Processor Arrays," *Abstracts of International Workshop on Hardware Fault Tolerance in Multiprocessors*, pp. 30-32, June 1989.
- [FiK85] A. L. Fisher and H.T. Kung, "Special Purpose VLSI Architectures: General Discussions and a Case Study," Prentice Hall, Englewood Cliffs, N. J., pp. 153-169, 1985.
- [FoK80] M. J. Foster and H. T. Kung, "The Design of Special-Purpose VLSI Chips," *Computer*, Vol. 13, No. 1, pp. 26-40, Jan. 1980.

- [GaH83] N. Gaitanis and C. Halatsis, "A new design method for m-out-of-n TSC checkers," *IEEE Transactions on Computers*, Vol. C-32, pp. 273-283, Mar. 1983.

- [HuM87] D. B. Hunt and P. N. Marinos, "A General Purpose Cache-Aided Error Recovery(CARER) Technique," *Proceedings of International Symposium on Fault Tolerant Computing*, pp. 170-175, 1987.

- [JhK90] Niraj K. Jha and S. Kundu, Testing and Reliable Design of CMOS Circuits, Kluwer Academic Publishers, 1990.

- [Ku82] H.T. Kung, "Why Systolic Architectures", *Computer*, pp. 37-46, Jan. 1982.

- [KuL87] S. Y. Kung, S. C. Lo, S. N. Jean, and J. N. Hwang, "Wavefront Array Processors- Concept to Implementation," *Computer*, pp. 18-33, July 1987.

- [KoD87] I. Koren and D. K. Pradhan, "Yield and Performance Enhancement through Redundancy in VLSI and WSI Multiprocessor Systems," *Proc. of the IEEE*, Vol. 74, No. 5, pp. 699-711, May 1986.

- [MaF78] M. A. Marouf and A. D. Friedman, "Design of self-checking checkers for Berger codes," *Proceedings of International Symposium on Fault Tolerant Computing*, Toulouse, pp. 179-184, June 1978.

- [MeC80] C. Mead and L. Conway, Introduction to VLSI Systems, Reading, MA: Addison-Wesley, 1980.

- [PaF82] Janak H. Patel and L. Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Transactions on Computers*, C-31, pp. 589-595.
- [PoB88] S. P. Popli and M. A. Bayoumi, "A Reconfigurable VLSI Array for Reliability and Yield Enhancement," *IEEE Proceedings of the International Conference on Systolic Arrays*, pp. 631-642, May 1988.
- [Pr80] D. K. Pradhan, "A new class of error correcting/detecting codes for fault-tolerant computer applications," *IEEE Transactions on Computers*, C-29, pp. 471-481, June 1980.
- [PrS80] D. K. Pradhan and J. J. Stiffler, "Error-correcting codes and self-checking circuits in fault-tolerant computers," *IEEE Computer*, pp. 27-37, March 1980.
- [Re74] S. M. Reddy, "A note on self-checking checkers," *IEEE Transactions on Computers*, Vol. C-23, pp. 1100-1102, Oct. 1974.
- [ReM78] D. Reynolds and G. Metze, "Fault detection capabilities of alternating logic," *IEEE Transactions on Computers*, Vol C-27, pp. 1093-1098, Dec. 1978.
- [SaS86] M. Sami and R. Stefanelli, "Reconfigurable Architectures for VLSI Processing Arrays," *Proceedings of the IEEE*, Vol. 74, No. 5, pp. 712-722, May 1986.

- [SiS82] D. P. Siewiorek and R. S. Swarz, The Theory and Practice of Reliable System Design. Bedford, MA: Digital, 1982.

- [SmM77] J. E. Smith and G. Metze, "The design of totally self-checking combinational circuits," *Proceedings of International Symposium on Fault Tolerant Computing*, pp. 130-134, June 1977.

- [SmL83] J. E. Smith and P. Lam, "A theory of totally self-checking system design," *IEEE Transactions on Computers*, Vol. C-32, pp. 831-843, Sept. 1983.

- [SuM77] I. E. Sutherland and C. Mead, "Microelectronics and computer science," *Scientific American*, Vol. 237, pp. 210-228, Sept. 1977.

- [YuT88] Yuval Tamir, Marc Tremblay, and David A. Rennels, "The Implementation and Application of Micro Rollback in Fault Tolerant VLSI Systems," *Proceedings of International Symposium on Fault-Tolerant Computing*, pp. 234-239, 1988.

- [Whi91] T. S. White, Distributed Control Reconfiguration Algorithms for 2-dimensional Mesh Architectures, Ph.D. Disserrtation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1991.

VITA

Vinay Murthy was born on April 20, 1968 in Kolar, India. He was awarded the degree of Bachelor of Technology in Electrical Engineering in May 1990 from Indian Institute of Technology, Bombay. After working for one year in BEL, India he entered the graduate program at Bradley Department of Electrical Engineering, Virginia Tech to pursue a Master of Science degree. Following the completion of his Master's degree, he will be accepting a position as Member Technical Staff with Hughes Network Systems in Germantown, Maryland.

A handwritten signature in black ink, appearing to read "Vinay" with a stylized flourish.