

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 2136851

**High yield and reliable sorting networks for VLSI and WSI
implementations**

Liang, Sheng-Chieh, Ph.D.

The University of Arizona, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

**HIGH YIELD AND RELIABLE SORTING NETWORKS
FOR VLSI AND WSI IMPLEMENTATIONS**

by

Sheng-Chieh Liang

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

WITH MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

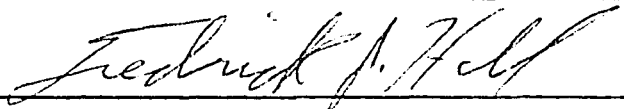
1 9 9 1

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

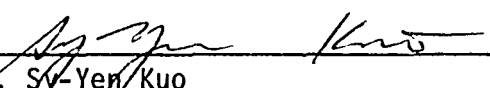
As members of the Final Examination Committee, we certify that we have read
the dissertation prepared by Sheng-Chiech Liang

entitled HIGH YIELD AND RELIABLE SORTING NETWORKS FOR VLSI AND WSI
IMPLEMENTATIONS

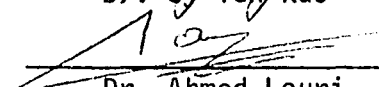
and recommend that it be accepted as fulfilling the dissertation requirement
for the Degree of Doctor of Philosophy.


Dr. Fredrick J. Hill

April 26, 1991
Date


Dr. Sy-Yen Kuo

April 26, 1991
Date


Dr. Ahmed Louri

April 26, 1991
Date

Date

Date

Final approval and acceptance of this dissertation is contingent upon the
candidate's submission of the final copy of the dissertation to the Graduate
College.

I hereby certify that I have read this dissertation prepared under my
direction and recommend that it be accepted as fulfilling the dissertation
requirement.


Dissertation Director
(Dr. Sy-Yen Kuo)

April 26, 1991
Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgement the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Sheng-Chieh Lang

ACKNOWLEDGMENTS

The author wishes to express his deepest gratitude to his dissertation advisor, Dr. Sy-Yen Kuo, for his support, encouragement, and insightful guidance throughout this dissertation work.

The author would also like to thank Dr. Fredrick J. Hill and Dr. Ahmed Louri for their advice and time, and all his colleagues in the Fault Tolerant Computing Group for their friendship and intellectual stimulation. The support by the National Science Foundation under Grant MIP-89-08648 is also gratefully acknowledged.

The author wishes to thank his parents and family whose love and support made it possible to complete this dissertation. Last but not least, he would like to express his appreciation to his wife, Huei-Chuan, for her continual encouragement, understanding, faith, and patience throughout his Ph.D. studies.

TABLE OF CONTENTS

LIST OF FIGURES	8
LIST OF TABLES	10
ABSTRACT	11
1. INTRODUCTION	13
1.1. Objective	13
1.2. Overview of the Fault-Tolerant Systolic Sorting Arrays	16
1.3. Overview of the Defect-Tolerant WSI Sorting Networks	16
1.4. Overview of the Trapezoid Sort	17
1.5. Overview of the Kth Smallest Value Extraction	18
1.6. Overview of the Modified Odd-Even Merge Procedure	18
2. THE FAULT-TOLERANT SYSTOLIC SORTING ARRAYS	19
2.1 Introduction	19
2.2 Array Architecture and Cell Realization	22
2.3. Properties of the Sorting Array	26
2.4. Fault Tolerance	34
2.5. Design of Totally Self-Checking Checkers	42
2.6. Evaluation and Discussion	47

2.7. Summary	52
3. THE DEFECT-TOLERANT WSI SORTING NETWORKS	54
3.1. Introduction	54
3.2. Hierarchical Modular Sorting Networks	56
3.3. Optimal Decomposition	61
3.4. Easily Reconfigurable Equivalent Networks	68
3.5. Defect Tolerance	72
3.6. Yield Analysis	79
3.7. Summary	84
4. THE TRAPEZOID SORT	86
4.1. Introduction	86
4.2. The Trapezoid Sort	87
4.3. Analysis and Time Complexity	90
4.4. Summary	98
5. THE Kth SMALLEST VALUE EXTRACTION	99
5.1. Introduction	99
5.2. Properties of the Trapezoid Sort	100
5.3. Finding the Kth Smallest Element	109
5.4. Summary	121
6. THE MODIFIED ODD-EVEN MERGE PROCEDURE	122
6.1. Introduction	122
6.2. The Modified Odd-Even Merge	123

6.3. Analysis	135
6.4. Summary	137
7. CONCLUSIONS	138
7.1. Summary and Discussions	138
7.2. Suggested Future Research	141
APPENDIX A: NETWORK TRANSFORMATION	142
APPENDIX B: PERMUTATION TRANSFORMATION	144
REFERENCES	147

LIST OF FIGURES

2.1. Odd-even transposition sort: comparator-network representation.	23
2.2. Odd-even transposition sort: word-level structure.	23
2.3. Structure of a <i>CS</i> element and the matrix of data registers.	25
2.4. Structure of a compare-swap cell and its logical functions.	26
2.5. Example errors.	28
2.6. A comparator-network representation of a faulty element at stage k	32
2.7. On-line fault diagnosis procedure.	38
2.8. Compare-swap element with bypass registers and switches.	41
2.9. System degradation scheme.	41
2.10. Totally self-checking data error detector.	43
2.11. A complete fault-tolerant sorting array with $N+3$ stages.	48
3.1. Hierarchical sorter.	57
3.2. The balanced sorter.	60
3.3. Cost C vs. p_1 for cases 1, 2, 3 and 4 with $p = 100$ and 105.	67
3.4. Omega network and modified data manipulator.	69
3.5. Shuffle permutation σ and Banyan permutation r	70
3.6. Balanced permutation τ and permutation ψ	71

3.7. Example reconfigurable structure of the odd-even transposition sorter.	72
3.8. Output functions of a switching element in the top-level.	73
3.9. On-line reconfigurable odd-even transposition sorter.	74
3.10. Example reconfiguration in the middle-level.	76
3.11. Butterfly interconnections with and without wrap-around wires.	78
3.12. Example yield analysis.	83
4.1. An example of snake-like row major indexing scheme.	88
4.2. The trapezoid sort algorithm.	89
4.3. An example output of step 2 with $n=7$	90
4.4. The maximum number of dirty rows = 3 for $Q_{7 \times 7}$	93
6.1. A compare-and-swap step in the column-sort.	124
6.2. A modified compare-and-swap step in the column-sort.	126
6.3. An example of the modified odd-even merge.	127
6.4. An example 8×8 L_A	128
6.5. Merge sort procedure.	131
6.6. Modified odd-even merge procedure.	133
6.7. An example of sorting $4N$ inputs.	134

LIST OF TABLES

2.1. Reservation table.	39
2.2. Overhead ratios.	51
3.1. The amount of redundancy for each case.	63
3.2. The cost C with $p=20$	64
3.3. The cost C with $p=100$	64
3.4. The cost C with $p=105$	65
3.5. The cost C with $p=200$	65
3.6. Regions covered by CRA and CRB	74
5.1. An example output of step 2 with $n=7$	101
5.2. The maximum number of dirty rows = 3 for $Q_{7 \times 7}$	104
5.3. The relationship between r_{p+k} and k	114
5.4. The distribution of zeros in each row after the second row sort.	115

ABSTRACT

In this dissertation, a novel approach to on-line error detection and correction for high throughput VLSI sorting arrays is presented first. Two-level pipelining is employed in the design which makes the proposed VLSI sorting array very efficient and suitable for real-time applications. In addition, all the checkers are designed as totally self-checking circuits such that the resulting sorting array is extremely reliable.

Next, in order to overcome the yield problem in WSI implementations a novel hierarchical modular sorting network is presented. This design is based on the tradeoffs in area and time between the odd-even transposition sort and the bitonic sort. More regularly structured equivalent sorting networks are introduced by replacing shuffle interconnections in the original sorting network with easily reconfigurable interconnections. Redundancy is provided at every level of the hierarchy. Hierarchical reconfiguration is implemented by replacing the defective cells with spare cells at the bottom level first, and goes to the next higher level. Yield analysis is performed to demonstrate the effectiveness of our approach.

Efficient implementation of parallel sorting algorithms for mesh-connected processor arrays are also considered in this dissertation. The *trapezoid sort* which has the properties of very simple control hardware and ease of implementation for mesh-connected processor arrays is developed. This algorithm is a combination of recursively sorting elements of two neighbor rows into opposite directions, sorting elements in each column, and a *cyclic shift* after the first row sort to rearrange the output order of each row. Its advantage is that the number of iterations is improved significantly compared with the existing parallel sorting algorithms on mesh-

connected processor arrays.

Based on this algorithm, an efficient method is proposed to find the median value of the input elements. The elements outside the boundary are excluded from the remaining sorting process to reduce the time complexity and the median value can be found without completely sorting the array. This method is then extended to finding the k th smallest element in the input array.

Finally, if the number of elements to be sorted is larger than N , the *trapezoid sort* algorithm can not be applied directly. Therefore, a *modified odd-even merge procedure* is presented to merge m sorted input sets. The modified odd-even merge procedure can sort two sets of data inputs concurrently by utilizing the idle processors and then merge them together. A speedup of $O(\log_2 m)$ over the previous merge-split method is achieved.

CHAPTER 1.

INTRODUCTION

1.1. Objective

Many applications in real-time digital signal and image processing need a high performance and special purpose architecture for parallel sorting on a huge amount of input data [1, 2, 3]. Sorting arrays which consist of a number of identical processing elements with regular interconnections and high concurrency factors [4], such as the odd-even transposition sort [5], the bitonic sort [6], and the perfect shuffle sort [7], are good candidates for real-time applications. Use of these arrays has become attractive mainly due to the availability of VLSI and WSI technologies at a reasonable cost.

Studies by Kung [8] indicate that both regular cell structures and simple interconnections will dominate the cost in VLSI or WSI implementations. Also by considering the ratio (A_w/A_t) of the wiring space to the total area as a function of the number of inputs, Horiguchi [9, 10] showed that the A_w/A_t ratio is approximately one for the perfect shuffle sort or the bitonic sort and is a constant 0.1 for the mesh connected odd-even sort when the number of inputs N is large.

Therefore, although both the perfect shuffle sort and the bitonic sort use less sorting elements ($O(N \log_2^2 N)$) than the odd-even transposition sort ($O(N^2)$) [11], the wiring complexities of the first two sorters make them more costly to implement than the odd-even sort since, for

large N , the wiring space will dominate the silicon area. This is why the more regularly structured odd-even transposition sort is a better candidate for VLSI implementations than other parallel sorting algorithms.

In addition to the area-time performance, reliability, availability, and continuous operation are important in real-time applications [12, 13]. Also, the defect tolerance capability (yield), the capability of a system to survive from defects, is also very important in manufacturing. In order to increase the system reliability and availability, a highly reliable sorting array which can detect multiple errors and correct a single error for on-line applications is presented. Furthermore, due to the large area and the processing technology limitation, defects seems unavoidable in VLSI and WSI implementation. Therefore, the sorting array needs to have defect tolerance capabilities. In this dissertation, we also present a novel hierarchical modular sorting network (*HMSN*) which is based on the tradeoffs between the simple communication scheme of the odd-even transposition sort and the fast convergent speed of the bitonic sort. Spare sorting elements are incorporated in every level of the hierarchy so that it can survive from defects in an efficient way.

Moreover, parallel sorting algorithms for two-dimensional mesh-connected processor arrays also have been intensively studied in [14,15,16,17] and more recently, in [18,19,20,21,22]. These earlier efforts were adaptations of inherently parallel algorithms such as the odd-even merge sort and the bitonic sort to the mesh-connected array in an efficient manner such that the time complexity is $O(n)$. However, these implementations spend most of the time in routing data to appropriate processors, and the complicated data movements in successive iterations result in complicated control structures and thus, offset the advantage of simple interconnections.

Recently, Sado and Igarashi [19], and Scherson and Sen [21] presented two similar parallel sorting algorithms independently, the parallel bubble sort and the shear sort, respectively, for two-dimensional SIMD model. These sorting algorithms are based upon a repeated application of the bubble-sort method [5] to the rows and columns of the array to be sorted. They are indeed two-dimensional sorting techniques and have the advantages that it is extremely simple to implement them in any of the two-dimensional computing models and their control complexity is reduced considerably due to their repetitive and nonrecursive nature. However, they have a drawback that $\lceil \log_2 n \rceil + 1$ iterations are required to sort an $n \times n$ input array.

In this dissertation, we present a new two-dimensional sorting algorithm, the *trapezoid sort*, which preserves the properties of simple control hardware and ease of implementation of the *row-column sort*, and the complexity is improved to $\lceil \log_2 l \rceil + 1$ iterations with $l \approx \sqrt{n}$. This algorithm can be used to find the k th smallest value of the inputs without the input sequence being completely sorted. Reduction on processing steps also means reduction in silicon area when the algorithm is implemented as a VLSI sorting network. However, the algorithms discussed above were designed to sort $N = n \times n$ inputs only, where N is the number of processors in the mesh array. If the number of elements to be sorted is larger than N , they can not be applied directly. To overcome this, the method in [14, 21] uses the merge-split operation to replace the compare-interchange (or compare-and-swap in this paper) operation and $O(m \log_2 m) T_N$ time complexity is required to sort mN inputs where T_N represents the time complexity to sort N inputs. Although that method is simple, it is not efficient. A novel *modified odd-even merge* method is proposed in this dissertation which can merge m sorted sets in $O(\frac{m}{2} \cdot \log_2 m) n$ time complexity. The other advantage of the proposed method is that it is quite simple and regular.

1.2. Overview of the Fault-Tolerant Systolic Sorting Arrays

In chapter 2, a novel approach to on-line error detection and correction for high throughput VLSI sorting arrays is presented. The error model is defined at the sorting element level and errors generated are considered as functional errors if the outputs from a faulty sorting element are not ordered correctly and as data errors if the output data values were modified by the faulty sorting element. Functional errors are detected and corrected by exploiting inherent properties as well as discovered special properties of the sorting array. Coding techniques and an on-line fault diagnosis procedure are developed to locate data errors. All the checkers are designed to be totally self-checking and hence the sorting array is highly reliable. Two-level pipelining is employed in our design which makes it very efficient and suitable for real-time application. The hardware overhead is not significant for typical array sizes and the time penalty is only 3 clock cycles. The structure is very regular and therefore, is very attractive for VLSI implementation.

1.3. Overview of the Defect-Tolerant WSI Sorting Networks

In order to overcome the yield problem in WSI, a novel hierarchical modular sorting network is presented in chapter 3. The design is based on the tradeoffs in area and time between the odd-even transposition sort and the bitonic sort. It uses less hardware than a single-level odd-even transposition sorter and reduces the wire complexity problem of the bitonic sorter in VLSI or WSI (wafer scale integration) implementation. The optimal number of levels in the hierarchy is analyzed and sorting capability of each level is derived to minimize the hardware complexity. More regularly structured equivalent sorting networks are introduced by replacing

shuffle interconnections in the original sorting network with easily reconfigurable interconnections. The hierarchical sorting network is very regular in structure after the equivalent network transformation and hence easier to include defect tolerance capability than any existing sorting network with the same time complexity. Redundancy is provided at every level of the hierarchy. Hierarchical reconfiguration is implemented by replacing the defective cells with spare cells at the bottom level first, and goes to the next higher level to perform reconfiguration if there is not enough redundancy at the current level. Yield analysis is performed to demonstrate the effectiveness of our approach.

1.4. Overview of the Trapezoid Sort

A parallel sorting algorithm, the *trapezoid sort*, for mesh-connected processor arrays is presented in chapter 4. Given a sequence of numbers mapped onto an $n \times n$ array, the *trapezoid sort* will generate a sorted output sequence stored in the array in *snake-like* row major order. This algorithm is a combination of recursively sorting elements of two neighbor rows into opposite directions, sorting elements in each column, and a *cyclic shift* after the first row sort to rearrange the output order of each row. It preserves the properties of very simple control hardware and ease of implementation, and has the advantage that the number of iterations improved significantly from $\lceil \log_2 n \rceil + 1$ to $\lceil \log_2 l \rceil + 1$ with $(l^2 + l)/2 \leq n < (l^2 + 3l + 2)/2$ in comparison with the existing parallel sorting algorithms on mesh-connected processor arrays.

1.5. Overview of the Kth Smallest Value Extraction

Properties of the *trapezoid sort* are derived in chapter 5. The maximum distance boundary of an element in an array to be sorted after the second iteration of the trapezoid sort from its position in the final sorted output array is determined first. An efficient method is also developed in chapter 5 to find the median value of the input elements by exploiting the property that the boundary distance will be reduced by half after each successive iteration. The elements outside the boundary are excluded from the remaining sorting process which reduces the complexity and the median value can be found without completely sorting the array. This method is then extended to find the k th smallest element in the input array.

1.6. Overview of the Modified Odd-Even Merge Procedure

The row-column sort algorithms on mesh-connected processor arrays, such as the *parallel bubble sort* and the *shear sort*, have the properties of very simple control hardware and ease of implementation. However, these row-column sort algorithms are based on the odd-even transposition sort such that half of the processors are idle during each basic comparison-interchange step. In addition, they are designed to sort N inputs only, where N is the number of processors in the array. If the number of elements to be sorted is larger than N , the row-column sort algorithms can not be applied directly. Therefore, a modified odd-even merge procedure is presented in chapter 6 to sort two sets of data inputs concurrently by utilizing the idle processors and then merge them together. This procedure is further generalized to merge m sorted input sets ($m > 2$) where each set can be initially sorted by any algorithm. A speedup of $O(\log_2 m)$ over the previous merge-split method is achieved.

CHAPTER 2.

THE FAULT-TOLERANT SYSTOLIC SORTING ARRAYS

2.1 Introduction

Many applications in real-time digital signal and image processing need a high performance and special purpose architecture for parallel sorting on a huge amount of input data. Sorting arrays which consist of a number of identical processing elements with regular interconnections and high concurrency factors [4], such as the odd-even transposition sort [5], the bitonic sort [6], and the perfect shuffle sort [7], are good candidates for real-time applications. Use of these arrays has become attractive mainly due to the availability of VLSI and WSI technologies at a reasonable cost. Studies by Kung [8] indicate that both regular cell structures and simple interconnections will dominate the cost in VLSI or WSI implementations. Also by considering the ratio (A_w/A_t) of the wiring space to the total area as a function of the number of inputs, Horiguchi [10] showed that the A_w/A_t ratio is approximately one for the perfect shuffle sort or the bitonic sort and is a constant 0.1 for the mesh connected odd-even sort when the number of inputs N is large. Therefore, although both the perfect shuffle sort and the bitonic sort use less sorting elements ($O(N \log_2^2 N)$) than the odd-even transposition sort ($O(N^2)$), the wiring complexities of the first two sorters make them more costly to implement than the odd-even sort since, for large N , the wiring space will dominate the silicon area. This is why the more regularly structured odd-even transposition sort is a better candidate for VLSI implementation than other parallel sorting algorithms.

Reliability, availability, and continuous operation are also very important in real-time applications. On-line error detection is the first requirement to increase the reliability. In order to increase the system availability, off-line diagnosis after on-line error detection should be avoided and the system should be able to automate the recovery process. In this chapter, we present a highly reliable sorting array which can detect multiple errors and correct a single error for on-line applications. In addition, it is highly available. As a systolic sorting array based on the odd-even transposition sort, it has a regular structure and simple interconnection links. Both the regularity and the simplicity are preserved by the presented fault tolerance technique so that redundancy can be included into the system easily, either to enhance the system performance or to replace the faulty elements. Also, it can be reconfigured easily to tolerate the faulty sorting elements located by the on-line fault diagnosis procedure and can be degraded gracefully after redundancy is exhausted.

Recently, an algorithm-based fault-tolerant sorter was proposed in [23]. They developed an on-line error detection method for the systolic priority queue [24] by applying the time redundancy approach to the operation of sorting a sequence of N inputs serially. Since it is a serial sorter which uses $N/2$ sorting elements and $2N$ clock cycles to sort N inputs, it is not suitable for real-time applications. Therefore, its entire structure as well as the fault tolerance techniques are different from the proposed highly pipelined sorting array. Also, by comparing the AT^2 complexities of the odd-even transposition sort with the systolic priority queue, it is found that the former is more cost effective than the latter by a factor of 2.

Extra cost incurred by bringing in fault tolerance features is minimized by exploiting the inherent properties of the embedded sorting algorithm. Properties such as nondecreasingly or nonincreasingly ordered output sequence is used to check the functional correctness of the sort-

ing array. In contrast with assuming that a faulty sorting element will transmit its inputs to the outputs unchanged or a faulty element can be located by some external circuits and then bypassed as in [25,26] and [27], a faulty sorting element in our error model can either pass or swap data incorrectly. Also, we discovered an important robust property of the odd-even transposition sorting array in which a single error can be masked automatically and multiple errors can be detected concurrently without disturbing the normal circuit operation.

In addition to checking the order of the outputs, the code-preserving property in data manipulation is employed to check whether the output data has been modified. Errors which violate the code-preserving property can be detected by using an appropriate coding technique. Depending on how critical the applications are, the requirements of fault coverage as well as the corresponding coding techniques will be different. Three example coding techniques are evaluated and the results are shown in section 2.6.

The total overhead of the proposed approach based on our analysis is much lower than previous fault tolerance techniques for other pipelined array processors [28,29,30], even if the checkers in the array are designed to be totally self-checking to increase the reliability. From the analysis in section 2.6, the overhead ratio is approximately $(54 + c)/14N$ where c is a constant determined by the data error coverage requirement as well as the adopted coding technique for the array. For example, with the simple parity check code, c is equal to four and the overhead ratio is less than 10% if N is greater than 42.

2.2 Array Architecture and Cell Realization

In order to have a high performance system, the two-level pipelining technique [31] which is frequently used in sorting arrays to achieve very high throughput [32,33] is employed in the design here. In addition to the use of the pipelined odd-even transposition sort as the word-level structure (section A), the systolic data flow concept [8] is used for the bit-level pipelining (section B). In this chapter, we use the term *CS element* to represent a word-level compare-swap element and the term *cell* to represent a bit-level compare-swap element. Also, without loss of generality, we will assume that the sorted output sequence is in nonincreasing order.

A. Array Architecture

The word-level pipelines can be achieved by one of the parallel sorting algorithms such as the odd-even transposition sort, the bitonic sort, the perfect shuffle sort, or the balanced sort. Based on the Aw/At ratio as discussed in section 2.1, the simple and regular odd-even transposition sort is adopted. An example odd-even transposition sorter with $N=5$ (without loss of generality, N is assumed to be an odd number) represented by Knuth's [5] comparator-network representation is shown in Fig. 2.1. Horizontal lines represent data paths and vertical lines represent comparisons between data values. As shown in Fig. 2.1, the five inputs, 1, 2, 3, 4, and 5 are nonincreasingly ordered at the outputs as 5, 4, 3, 2, and 1. The word-level implementation of the systolic sorting array based on the odd-even transposition sort is shown in Fig. 2.2 where each vertical line in Fig. 2.1 is realized by a *CS element*. The parallel odd-even transposition sorting array consists of a cascade of N stages with $N(N-1)/2$ *CS elements* in each stage to sort N input data elements [5]. Each *CS element* in the sorting array compares two n -bit input numbers x and y and swaps these two values if $x < y$. Data registers (D) in Fig. 2.2 are used as

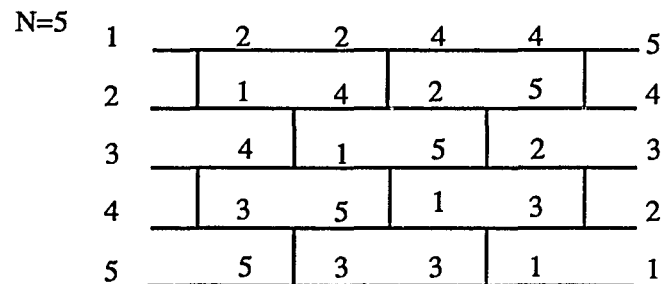


Figure 2.1. Odd-even transposition sort: comparator-network representation.

delay buffers so that input data sets can be synchronized by the system clock and pipelined through stages of the sorting array.

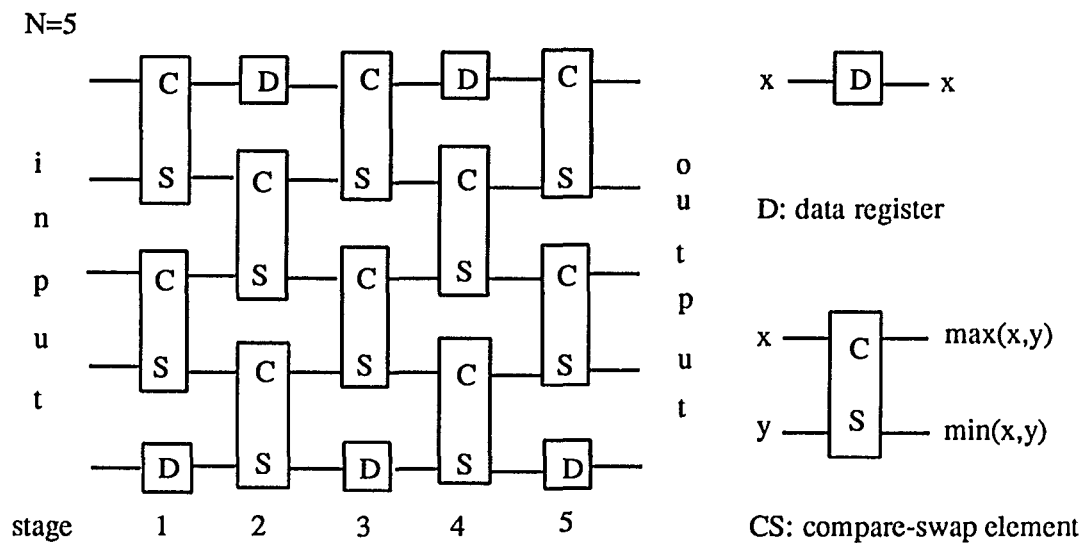


Figure 2.2. Odd-even transposition sort: word-level structure.

B. Cell Realization

In the word-level, there are only two types of elements in the sorting array: data registers (D) and compare-swap elements (CS). For each CS element, n bit-level comparisons are required to compare two n -bit binary numbers. These n steps of comparisons can be implemented by either a serial or a parallel method. Since the goal here is to have high throughput systems, the systolic data flow concept is also applied in the bit-level pipelines. A matrix of single-bit data registers (d) is cascaded before the input stage to synchronize the data flow as shown in Fig. 2.3. (Note that this matrix of d registers is required only before the CS elements of the first stage.) These data registers are arranged as a lower right triangular matrix such that input data bits can enter the systolic sorting array in a skewed fashion. That is, for a CS element, when c_n has finished processing x_n and y_n the comparison results can pass to cell c_{n-1} together with the two inputs x_{n-1} and y_{n-1} at the same time. Therefore, c_n can process the next inputs a_n and b_n when c_{n-1} is processing x_{n-1} and y_{n-1} . The n cells of each CS element are chained together by the swap control lines r and s .

Logical operations of a cell c_i in a CS element are described in the following: (1) Signals s_{i+1} and r_{i+1} from cell c_{i+1} to cell c_i indicate whether any of the more significant bits than bit i has been swapped or not. (2) Inputs x_i and y_i are the i th bits of the two input words x and y to a CS element, respectively. (3) The signal r_i indicates whether $(x_n, \dots, x_i) = (y_n, \dots, y_i)$ or not ($r_i = 0$ or 1 , respectively) and s_i indicates whether $(x_n, \dots, x_i) < (y_n, \dots, y_i)$ or not ($s_i = 1$ or 0 , respectively). (4) α_i and β_i are two output data bits from cell c_i with $\alpha_i \geq \beta_i$. Therefore, we have the bit-level cell structure and logical equations as shown in Fig. 2.4.

In the case of $s_{i+1} = 1$, it means $(y_n, \dots, y_{i+1}) > (x_n, \dots, x_{i+1})$. Both s_i and r_i should then be set to 1 and passed to cell c_{i-1} to indicate that $x < y$ and cell c_i should swap the two input bits

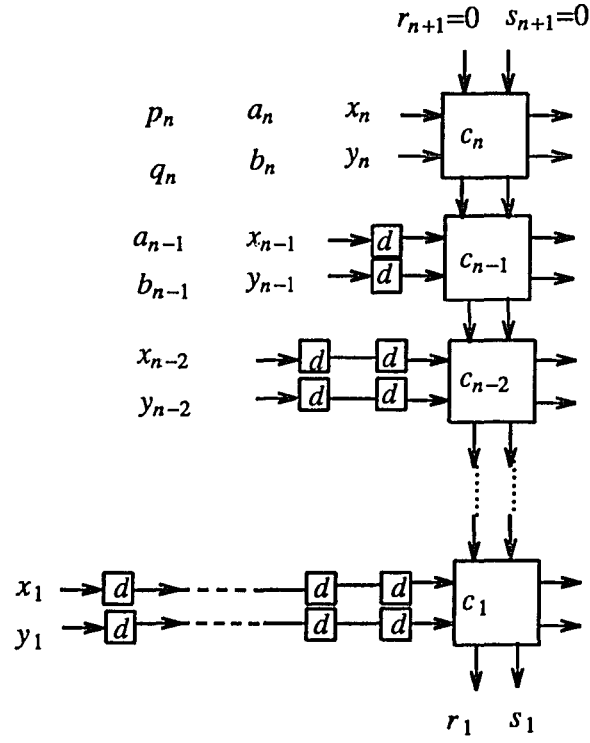


Figure 2.3. Structure of a CS element and the matrix of data registers.

x_i and y_i . Otherwise, if $s_{i+1} = 0$ and $r_{i+1} = 1$, $(y_n, \dots, y_{i+1}) < (x_n, \dots, x_{i+1})$ and (s_i, r_i) should be $(0, 1)$, i.e., no swap is needed. The case in which both r_{i+1} and s_{i+1} are zeros represents $(y_n, \dots, y_{i+1}) = (x_n, \dots, x_{i+1})$ and the order of x and y will be determined by x_i and y_i in the following three cases: (1) If $x_i = y_i$, we have $s_i = r_i = 0$, $\alpha_i = x_i$, and $\beta_i = y_i$. (2) If $x_i < y_i$, then $s_i = r_i = 1$, $\alpha_i = y_i$ and $\beta_i = x_i$. (3) Otherwise, $s_i = 0$, $r_i = 1$, $\alpha_i = x_i$, and $\beta_i = y_i$.

For example, let $x = (x_n, \dots, x_1)$, $y = (y_n, \dots, y_1)$ be two inputs of a CS element in the first stage. x_n and y_n are processed in cell c_n of the CS element first. Initially $r_{n+1}=0$ and $s_{n+1}=0$. After the comparison of x_n and y_n in cell c_n , the swap control signals s_n and r_n from c_n will be passed to c_{n-1} . At the next clock, x_{n-1} and y_{n-1} together with s_n and r_n are processed in c_{n-1} , another input data set a_n and b_n is processed at c_n at the same time.

The swap control signals s_1, r_1 from cell c_1 indicate whether the two inputs, x and y , have been swapped ($s_1 = 1, r_1 = 1$) or not ($s_1 = 0$). We call s_1 the *swap-indicator* since it alone can tell us if there is any swap operation performed in the corresponding stage.

2.3. Properties of the Sorting Array

In order to evaluate the fault tolerance techniques, the error model which describes the effect of physical faults on the sorting element will be defined first in subsection A. The error model defined here is quite general that it can cover many faults whose nature are not apparent. Properties of the sorting array which will be exploited to introduce fault tolerance capabilities are then derived in subsection B.

A. Error Model

The error model is defined at the *CS* element level. A *CS* element which contains physical faults can generate errors such as swapping its inputs incorrectly, modifying the data values, or both, and can be classified as a *functional error*, a *data error*, or a *hybrid error*, respectively.

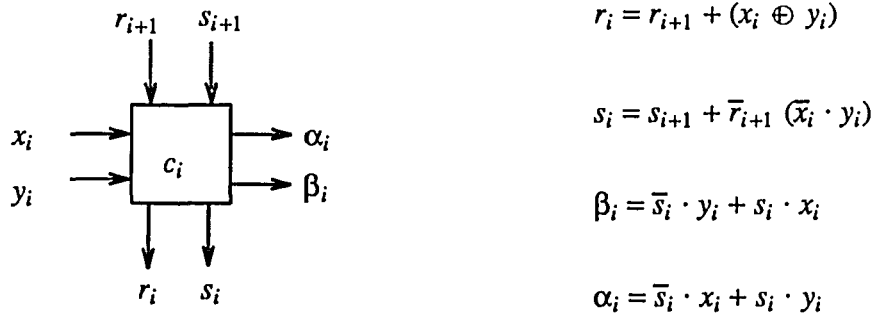


Figure 2.4. Structure of a compare-swap cell and its logical functions.

For example, stuck-at faults on the two swap control lines can cause functional errors and stuck-at faults on the communication links can cause data errors. Effect of faults on links between stages i and $i+1$ is lumped into stage $i+1$ such that errors in communication links are also representable in this word-level error model. Faults in communication links are less common [34] but more severe since, in a sorting array, a faulty communication link will cause the entire output data useless unless a reconfiguration process followed by a recovery process is applied.

Example errors are shown in Fig. 2.5 where x and y represent four-bit numbers. The example in Fig. 2.5(a) shows that the element with faulty control lines performs an incorrect swap and thus represents a functional error. An example data error is shown in Fig. 2.5(b) which indicates that the value y has been modified. A hybrid error is shown in Fig. 2.5(c). A CS element with a hybrid error will generate both incorrect order and data values at the outputs and is regarded as having multiple errors.

B. Properties

The first property is that the systolic sorting array based on the odd-even transposition sort with N stages and $(N-1)/2$ elements in each stage is a valid sorting array and a random input sequence will be correctly ordered at the outputs [35]. The second property is that the sorting array is a code-preserving sorter. This is due to the fact that the sorting array consists of CS elements and data registers only, no logical or arithmetic operation which can modify data values is performed during normal circuit operations. Therefore, the order of input sequence may be modified at the outputs but the coded input values should be preserved.

These two properties inherently exist in all sorting arrays and any sorter can be examined functionally according to these two properties. In addition to these two common properties, we

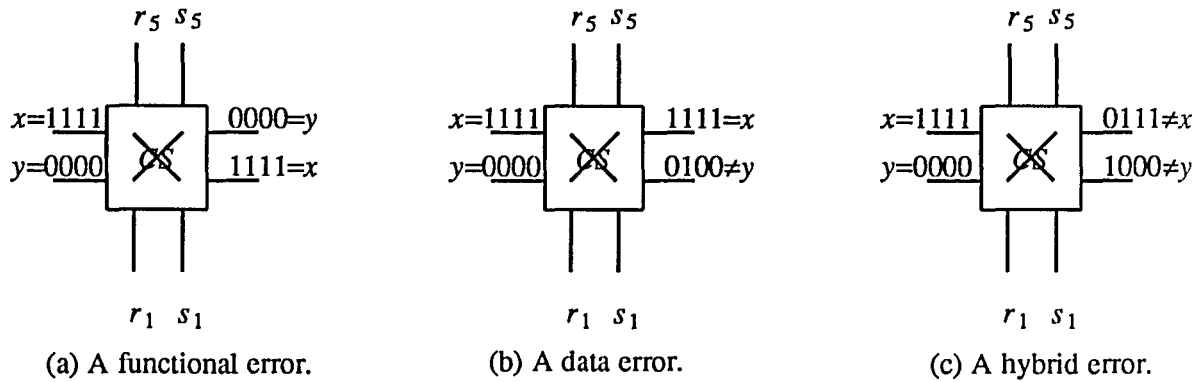


Figure 2.5. Example errors.

derive a special property for functional error checking which can be applied to all pipelined sorting algorithms and a robust property for the odd-even transposition sort only, in which any single functional error can be recovered automatically. Based on these two special properties, the sorting array can be designed with high reliability and low overhead.

It should be noted that as was discussed in subsection A, it is possible for a CS element to swap its inputs incorrectly such that the entire output sequence from a sorting array is not nonincreasingly ordered. From the comparator-network representation of the odd-even transposition sort (Fig. 2.1), we can see that two neighbor stages in the odd-even transposition sorting array completely compare all pairs of adjacent inputs in two clock cycles. Therefore, if two additional neighbor stages which include an odd-numbered stage and an even-numbered stage are added after the last stage of any sorting array, they can be used as a checker to check whether the outputs from the sorting array are ordered or not. If the output sequence is correctly ordered, no swap operation will be executed in any CS element of these two additional stages, otherwise, some of these CS elements will perform swap operations and it represents that the output sequence from the sorting array is not correctly ordered. We call these two

stages the *Nonincreasing-Order-Checker (NOC)*, and the detailed proof will be presented in Theorem 2.1.

Let V_1 represent the input vector of N values to the sorting array (*i.e.*, the input of the first stage) and V_h represent the input vector of the h th stage. $V_h[i]$ represent the value located at the i th input line of stage h .

THEOREM 2.1: The nonincreasing-order-checker (*NOC*) can determine whether the output sequence from the sorting array is correctly ordered or not.

PROOF: The outputs of the sorting array at the N th stage will be nonincreasingly ordered such that

$$V_{N+1}[1] \geq V_{N+1}[2] \geq V_{N+1}[3] \geq \dots \geq V_{N+1}[N]$$

if there is no faulty *CS* element in the array. The *CS* elements in stages $N+1$ and $N+2$ of the *NOC* will then compare $V_{N+1}[i]$ to $V_{N+1}[i+1]$, $i=1$ to $N-1$, and should not perform any swap operation. If the output from the *NOC* which is the *ORing* of the *swap-indicators* in these two stages is set to 1, it means that at least one of the *CS* elements of the *NOC* has made a swap operation and therefore, the output sequence from stage N is not correctly ordered. \square

The second property is the robust property of the odd-even transposition sort. This robust property is very important in on-line real-time applications. For on-line applications, the probability of a single error is much higher than multiple errors. If a single error can be recovered automatically without interrupting the entire system, the system availability will be increased significantly. Before we prove this property, two variables $ex(i,h)$ and $bp(i,h)$ are introduced first in Definition 2.1 to represent the number of exchanges and bypasses executed, respectively, if we compare $V_h[i]$ with $V_h[j]$, for all $j > i$.

DEFINITION 2.1: Let $ex(i, h) = \sum_{j=1}^{N-i} c(V_h[i]:V_h[i+j])$ where $c(V_h[i]:V_h[i+j])=1$ if

$V_h[i] < V_h[i+j]$, and $c(V_h[i]:V_h[i+j])=0$, otherwise. Similarly, let $bp(i, h) = \sum_{j=1}^{N-i} \bar{c}(V_h[i]:V_h[i+j])$

where $\bar{c}(V_h[i]:V_h[i+j])=1$ if $V_h[i] \geq V_h[i+j]$ and $\bar{c}(V_h[i]:V_h[i+j])=0$, otherwise. □

LEMMA 2.1: If $\sum_{i=1}^{N-1} ex(i, h)=0$, then the input sequence to stage h is in nonincreasing order

after being processed by stages 1, 2, ..., $h-1$.

PROOF: Since $ex(i, h) \geq 0$, $\sum_{i=1}^{N-1} ex(i, h)=0$ means that $ex(i, h)=0$ for all $i=1$ to $N-1$. There-

fore, $V_h[1] \geq V_h[i]$ (for $1 < i \leq N$) since $ex(1, h)=0$, and $V_h[2] \geq V_h[i]$ (for $2 < i \leq N$) since $ex(2, h)=0$, and so on . . ., $V_h[N-1] \geq V_h[N]$ since $ex(N-1, h)=0$. That is, $V_h[1] \geq V_h[2] \geq \dots \geq V_h[N]$. □

LEMMA 2.2: $ex(i, h) + bp(i, h) = N - i$ and $\sum_{i=1}^{N-1} ex(i, h) + \sum_{i=1}^{N-1} bp(i, h) = \binom{N}{2} = \frac{N(N-1)}{2}$

PROOF: This lemma can be proved by assuming that a *bubble sort* is applied to the input vector of stage h . Thus, the values of $ex(i, h)$ and $bp(i, h)$ can be viewed as the number of exchanges and bypasses, respectively, required to move the value at line i to its final position. Hence, $ex(i, h) + bp(i, h) = N - i$ and the total number of operations required by the bubble sort to

sort the corresponding input vector is equal to $\sum_{i=1}^{N-1} N - i = \sum_{i=1}^{N-1} ex(i, h) + \sum_{i=1}^{N-1} bp(i, h) = \binom{N}{2} = \frac{N(N-1)}{2}$. □

In the following analysis, we will assume that a functional error is generated by the CS element in stage k ($k \leq N$) which compares two inputs on lines x and $x+1$ (see Fig. 2.6).

LEMMA 2.3: For $i \neq x$, if $V_k[i] < V_k[i+1]$ then $ex(i, k+1) = ex(i, k) - 1$ else $ex(i, k) = ex(i, k+1)$. For $i = x$, if $V_k[x] \geq V_k[x+1]$ then $ex(x, k+1) = ex(x, k) + 1$ else $ex(x, k) = ex(x, k+1)$.

PROOF: For $i \neq x$, $ex(i, k) = \sum_{j=1}^{N-i} c(V_k[i]:V_k[i+j])$ and $ex(i, k+1) = \sum_{j=1}^{N-i} c(V_{k+1}[i]:V_{k+1}[i+j])$.

From these two functions we can see that the difference between the results of $ex(i, k)$ and $ex(i, k+1)$ will depend on the values of $V_k[i]$, $V_k[i+1]$ and the operation of the CS element between the two lines i and $i+1$. Other CS elements in stage k will not affect the relationship between $ex(i, k)$ and $ex(i, k+1)$. If $V_k[i] < V_k[i+1]$ and there is a CS element between lines i and $i+1$ then this CS element will perform an exchange operation on the two input values at the outputs. So we have $V_{k+1}[i](=V_k[i+1]) > V_{k+1}[i+1](=V_k[i])$, i.e., $ex(i, k+1) = ex(i, k) - 1$. Otherwise, $ex(i, k) = ex(i, k+1)$, for $k \leq N$.

For $i = x$, as the analysis in the above, the difference between $ex(x, k)$ and $ex(x, k+1)$ depends on the values of $V_k[x]$, $V_k[x+1]$ and the operation of the CS element between the two lines x and $x+1$. According to the outputs of the faulty element, this functional error can be classified as either incorrect swapping or incorrect bypassing and the corresponding relationship between $ex(x, k)$ and $ex(x, k+1)$ will be $ex(x, k) + 1 = ex(x, k+1)$ for making an incorrect swapping or $ex(x, k) = ex(x, k+1)$ for making an incorrect bypassing, respectively. □

From the analysis of Lemma 2.3, we know that the number of exchanges required to move the value at line i of stage k to its final position is not affected by other CS elements that do not compare the two values $V_k[i]$ and $V_k[i+1]$. That is the faulty CS element in stage k will not increase the number of exchanges required to move the value at line i to its final position unless $i = x$.

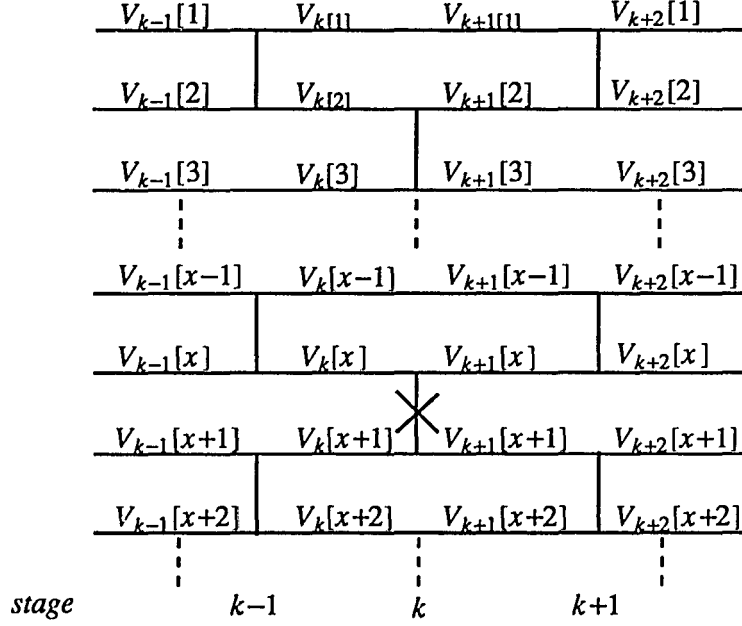


Figure 2.6. A comparator-network representation of a faulty element at stage k .

THEOREM 2.2: The systolic sorting array for N inputs based on the odd-even transposition sort with $N+2$ stages can recover from a single functional error in the first N stages automatically.

PROOF: Depending on the number of exchanges performed by the fault-free CS elements in stage k and the results derived in Lemma 2.3, we have the following two cases:

$$(1) \sum_{i=1}^{N-1} ex(i, k) + 1 = \sum_{i=1}^{N-1} ex(i, k+1), \text{ that is, all the } CS \text{ elements in stage } k \text{ perform bypass}$$

operations except the faulty one which generates an incorrect swap. According to the configuration of the odd-even transposition sort (as shown in Fig. 2.6), we have $V_{k+1}[1] \geq V_{k+1}[2] \geq \dots \geq V_{k+1}[x]$ and $V_{k+1}[x+1] \geq V_{k+1}[x+2] \geq \dots \geq V_{k+1}[N]$. Therefore, CS elements in stages $k+1$ and $k+2$ will not perform any swap operation except the one that compares $V_{k+2}[x]$ and $V_{k+2}[x+1]$ and it corrects the result generated by the faulty element at stage k .

Therefore, $\sum_{i=1}^{N-1} ex(i, h) = 0$ for $h > k+2$ since $V_{k+3}[1] \geq V_{k+3}[2] \geq \dots \geq V_{k+3}[x]$,

$V_{k+3}[x+1] \geq V_{k+3}[x+2] \geq \dots \geq V_{k+3}[N]$, and $V_{k+3}[x] \geq V_{k+3}[x+1]$. Thus, input sequence to stage $k+3$ has been sorted and it will not change the order any further more in the sorting array from stages $k+3$ through $N+2$.

(2) $\sum_{i=1}^{N-1} ex(i, k) \geq \sum_{i=1}^{N-1} ex(i, k+1)$, that is, at least one *CS* element in stage k executes a swap

operation in addition to the faulty one. Let $ex_b(i, h)$ and $ex_f(i, h)$ be equivalent to $ex(i, h)$ and $bp_b(i, h)$ and $bp_f(i, h)$ be equivalent to $bp(i, h)$ except that the subscript "b" means that the faulty stage k is assumed bypassed and "f" means that the faulty stage is assumed not bypassed. In the following analysis, we will complete the proof by first showing that the number of exchanges required for the faulty stage k is bypassed will be equal to 0 at the input of stage $N+3$ (i.e., $ex_b(i, N+3) = 0$) and then showing that $ex_b(i, h) \geq ex_f(i, h)$ for $h > k$ so that we also have $ex_f(i, N+3) = 0$.

If stage k is bypassed, the function of stage $k-1$ will be duplicated by stage $k+1$ so that stage $k+1$ can be viewed as bypassed. Thus, the normal execution of stage k is performed by stage $k+2$ and the normal execution of stage $k+1$ is performed by stage $k+3$, ..., and the normal execution of N is performed by stage $N+2$. So the output sequence from stage $N+2$ is sorted.

From Lemma 2.1, we have $\sum_{i=1}^{N-1} ex_b(i, N+3) = 0$. By comparing $\sum_{i=1}^{N-1} ex_b(i, h)$ with $\sum_{i=1}^{N-1} ex_f(i, h)$ for

$k \leq h \leq N+3$, we can show that $\sum_{i=1}^{N-1} ex_b(i, h) \geq \sum_{i=1}^{N-1} ex_f(i, h)$. The reason is described in the following:

Due to the result from Lemma 2.2, we have $\sum_{i=1}^{N-1} ex(i, h) + \sum_{i=1}^{N-1} bp(i, h) = \binom{N}{2} = \frac{N(N-1)}{2}$. Since the

input vector V_k for assuming faulty stage k is either bypassed or not is the same. So that

$\sum_{i=1}^{N-1} bp_f(i, k) = \sum_{i=1}^{N-1} bp_b(i, k)$. Because the input vector V_k are processed by more functionally

normal *CS* elements before arriving stage h ($h > k$) in the case of faulty stage k is not bypassed than in the case of stage k is bypassed, therefore, we have $\sum_{i=1}^{N-1} bp_f(i, h) \geq \sum_{i=1}^{N-1} bp_b(i, h)$ and $\sum_{i=1}^{N-1} ex_b(i, h) \geq \sum_{i=1}^{N-1} ex_f(i, h)$ for $h \geq k$. Since $\sum_{i=1}^{N-1} ex_b(i, N+3) \geq \sum_{i=1}^{N-1} ex_f(i, N+3)$ and $\sum_{i=1}^{N-1} ex_b(i, N+3) = 0$, we have $\sum_{i=1}^{N-1} ex_f(i, N+3) = 0$. Again from Lemma 2.1, we know that the outputs of stage $N+2$ have been sorted and therefore, the single functional error is recovered by the two extra stages. \square

2.4. Fault Tolerance

Fault tolerance techniques such as recomputing in different stages or elements [36, 28, 37, 38], and recomputing with shifted operands [30] can detect errors in pipelined array processors, but the requirement of 100% time overhead is not tolerable in real-time applications. Therefore, we adopt the algorithm-based approach [39, 40, 41] to design a fault tolerant systolic sorting array with the capabilities of concurrent error detection and correction and minimize the overhead by using the properties we derived in the last section.

In subsection A, by checking the two general invariant properties, the sorting array has the capability of concurrent error detection. By exploiting the special property we derived in Theorem 2.2, the sorting array can correct a single functional error during the normal operation. It is difficult to correct data errors in a sorting array during the normal operation. Even if the faulty bits can be detected and corrected by some coding techniques [42, 43, 44] such as the Hamming code and the Berger code, [45] the output sequence is no longer correctly ordered. Therefore, with the assumption that the hardware used for off-line diagnosis and yield enhancement such as the multiplexers and the bypass registers in each *CS* element are fault-free, we will

present a fast on-line fault diagnosis procedure in subsection B to locate the faulty sorting elements. This assumption is appropriate because those additional circuits are usually included for the purpose of off-line testing and reconfiguration [31] in the manufacturing phase and they are fault free before operation and not activated during the normal on-line operation.

For mission-critical applications, the restart time should also be minimized and therefore, an efficient on-line reconfiguration procedure is presented in subsection C. The presented sorting array can also be degraded gracefully. As will be discussed in subsection C, it can be degraded to sort less input data and tolerate more faulty elements if it runs out of redundancy.

A. Concurrent Error Detection and Correction

As proved in Theorem 2.1, whether the output sequence is in nonincreasing order or not can be detected by the *NOC*. Error correction for a single functional error is done automatically as shown in Theorem 2.2. Therefore, the two stages added to a sorting array can be either a checker or a single error corrector. These two stages are sufficient for a single error. But for multiple errors, two more stages are required to detect other errors after the first error has been corrected by the first two added stages.

The problem of who will check the checkers is very important in mission critical applications. The two additional stages used to recover a single error in the array will not be able to recover errors in themselves. The errors in the *NOC* itself will generate a useless result if the *NOC* does not have a self-checking capability to check its own outputs. Therefore, from Theorems 2.1 and 2.2, the sorting array for N inputs can be implemented with $N+4$ stages for error detection and correction. The first N stages are for normal sorting functions. Stages $N+3$ and $N+4$ are used as the *NOC* and will be designed to be *totally self-checking (TSC)* [46]. (The details on implementing a *TSC* checker will be discussed in section 2.5). Stages $N+1$ and

$N+2$ which are used to correct a single functional error do not need to be *TSC* circuits since their outputs are checked by stages $N+3$ and $N+4$ (*NOC*). However, in the following theorem, we will show that stage $N+4$ can be omitted from the sorting array if stage $N+2$ is implemented by *TSC* circuits.

THEOREM 2.3: The systolic sorting array with a total of three additional stages, stages $N+1$, $N+2$ and $N+3$, where stages $N+2$ and $N+3$ are implemented by *TSC* circuits can tolerate one incorrect swap operation and check whether the output sequence is nonincreasingly ordered or not.

PROOF: From the properties derived in Theorems 2.1 and 2.2, a systolic sorting array with a total of four additional stages has the capability to tolerate one incorrect swap operation and to check whether the output sequence is ordered or not. In the following analysis, we will prove that if stages $N+2$ and $N+3$ are implemented with *TSC* circuits, the stages $N+1$ and $N+2$ can be used to correct a single functional error and stage $N+3$ itself can be used to detect multiple functional errors in the sorting array.

According to the properties of *TSC* circuits [46], the checkers implemented as *TSC* circuit are code disjoint, fault secure, and self-testing. If stage $N+2$, which compares $V_{N+2}[1]$ and $V_{N+2}[2]$, $V_{N+2}[3]$ and $V_{N+2}[4]$, ..., $V_{N+2}[N-2]$ and $V_{N+2}[N-1]$, is implemented as *TSC* circuits, we should have $V_{N+3}[1] \geq V_{N+3}[2]$, ..., $V_{N+3}[N-2] \geq V_{N+3}[N-1]$ at the output of stage $N+2$ due to the fault secure property, otherwise, errors in this stage will be detected by the stage itself. Therefore, if the output sequence is nonincreasingly ordered at the output of stage $N+2$, then stage $N+3$ which takes this sequence as input and compares $V_{N+3}[2]$ and $V_{N+3}[3]$, ..., $V_{N+3}[N-1]$ and $V_{N+3}[N]$, should not do any swap operation, otherwise, the output sequence is not ordered. This shows that stage $N+4$ can be omitted.

□

In addition to checking the correctness of the output order, we can check whether the input data values are preserved during the normal operation or not by using appropriate coding techniques. The choice of a data error detection method is very flexible depending on the properties of the sorting array, the type of errors to be detected, and the fault coverage requirement. Arithmetic codes such as residue code[47] have good fault coverage for all kinds of errors, but they require large devices. Coding techniques for error detection in communication lines such as Berger code and modified Berger code [48] require less area overhead but they are efficient in detecting unidirectional errors only. Fault coverages and overhead analysis for different options in coding techniques will be discussed in section 2.6.

B. On-Line Fault Diagnosis

We have proved (in Theorem 2.2) that whether the fault-free CS elements in the faulty stage k are bypassed or not, two extra stages are required any way to mask the effect of a faulty CS element in the faulty stage k ($k \leq N$), i.e., the fault-free CS element in the faulty stage are useless. Therefore, instead of locating individual faulty elements, the on-line fault diagnosis procedure only needs to identify the location of the entire faulty stage and this makes our reconfiguration procedure simpler.

The diagnosis procedure is described in Fig. 2.7. The input set I which generates the data error is reapplied repeatedly to diagnose the faulty stage. A *reservation table* which shows an example diagnosis for a sorting array with $N = 5$ and three extra stages is in Table 2.1 where *DED* represents a data error detector. A marked entry at the (n, m) th position of the table indicates that the stage $K = n$ will be activated m time units later after the initiation of the fault-diagnosis procedure. Empty spaces in the table represent idle stages that perform only bypass operation.

```

Procedure On_Line_Fault_Diagnosis;
begin
    roll back input data I;
    /* I is the inputs which generate the data error */
    k = 1;
    /* k is used to control which stages should be bypassed */
    for t: = 1 to N+3 do begin
        /* t represents the clock sequence */
        if t is odd then k = k+1;
        input I and bypass stages k to N+3;
    end;
    for t: = N+4 to 2(N+3) do begin
        if data error detector is set at t
            then stage t - (N+3) is faulty;
    end;
end. /* all stages process data normally except the stages
      specified to be bypassed */

```

Figure 2.7. On-line fault diagnosis procedure.

At $t = 1$, the input set I is applied and processed at stage 1 (we call it I_{11} to represent that it is reapplied into the sorting array at $t = 1$ and has been processed by stage 1) and will pass through all the other stages without being processed. The bypassing capability of a CS element and a stage will be discussed in the next section. At $t = 2$, the same input set I is applied again to stage 1 (we call it I_{21}) and I_{11} will not be processed until $t = 9$. That is, I_{11} bypasses stages 2 to 8 and is then checked by *DED*. At $t = 3$, the same input set I is applied again to stage 1 (we call it I_{31}), and at the same time I_{21} enters stage 2 (we call it I_{22}). After this, I_{22} will not be processed until it is checked by *DED* at $t = 10$. Thus, for $t=1, 2, \dots, N+3$, these reapplied inputs are processed at stage 1, stages 1 through 2, ..., stages 1 through $N+3$, respectively, such that the data set I which generates incorrect data at the outputs due to a faulty CS element in stage m will set the data error indicator at time $t = N+3+m$ during the on-line fault diagnosis.

Although this on-line fault diagnosis procedure is designed to locate the faulty stage which generates data errors, by adding some extra hardware this procedure can be adapted to locate the faulty element which generates either a functional error or a data error. Since each data word contains the check bits, if we can access these check bits we can identify the faulty sorting element in stage m when a data error is detected at $t=(N+3)+m$. To locate the faulty elements generating a functional error, both *swap-indicators* of stage $N+2$ and stage $N+3$ should be checked. These two stages are activated and bypassed alternately so that inputs at $t=1, 3, \dots, N$ are checked by stage $N+2$ and inputs at $t=2, 4, \dots, N+1$ are checked by stage $N+3$ respectively. If some of the sorting elements in stage m are faulty, the corresponding sorting elements in stage $N+2$ (m is odd) or $N+3$ (m is even) will set the *swap-indicators* at time $t=(N+3)+m$.

C. On-Line Reconfiguration and Performance Degradation

In order to achieve fast on-line reconfiguration, bypass registers (B) and multiplexers (*mux*) (see Fig. 2.8) which are usually added to a processing element for manufacturing reconfiguration [31] are utilized in our on-line reconfiguration. The multiplexer can be enabled by two control lines, row bypass (r) or column bypass (c). In the normal operation, (r, c) is reset as (0, 0) and outputs from the CS elements are selected. If a CS element is faulty, it is bypassed by setting either r or c to 1. In order to reduce the cost of bypass control circuitry, only the word-level control scheme is considered as shown in Fig. 2.9 where a CS element is bypassed when either the corresponding bit in the row bypass control register or the column bypass control register is set to 1. All bypass control lines of cells in the same CS element are set or reset together. Each bit of the row bypass control register controls all CS elements in the same row and each bit of the column bypass control register controls all CS elements in the same column (stage). The entire sorting array can also be degraded to sort $N-1$, $N-2$ or less

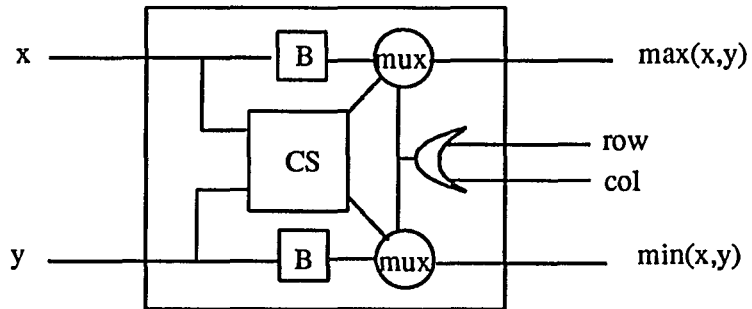


Figure 2.8. Compare-swap element with bypass registers and switches.

inputs as shown in Fig. 2.9. Two dotted lines in row 4 and stage 5 of Fig. 2.9 means that *CS* elements are bypassed in the corresponding row and column and the sorting array can then be used to sort $N-1$ inputs.

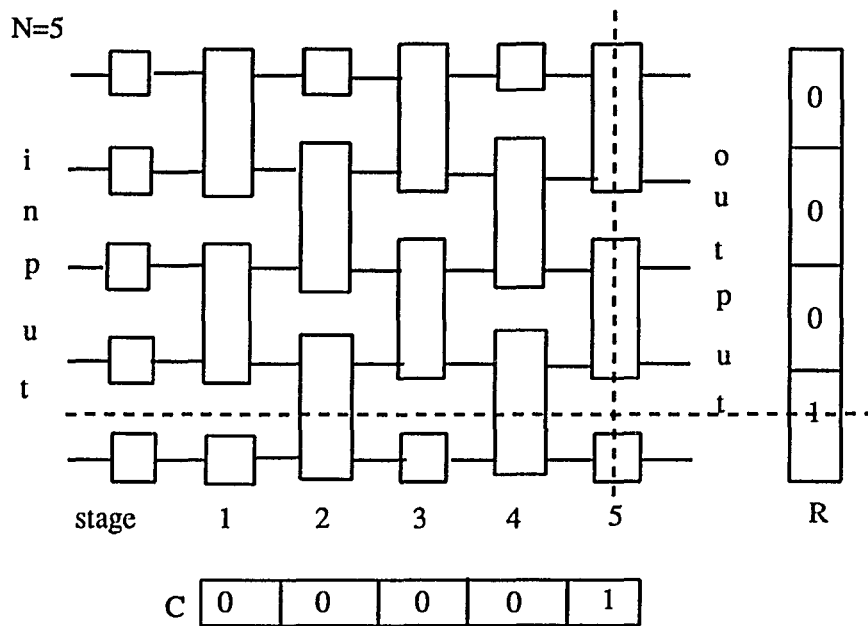


Figure 2.9. System degradation scheme.

2.5. Design of Totally Self-Checking Checkers

We have designed a data error detector and an *NOC* to detect data errors and functional errors, respectively, in the last section. It is always desirable to design checkers which can detect errors in the checker itself as well as in its inputs. This leads us to design checkers which are totally self-checking (*TSC*). The concept of a totally self-checking checker has been introduced in [49] as a circuit which is fault secure, self-testing, and code disjoint [46].

A. Design of a Totally Self-Checking Data Error Detector

A general structure of the totally self-checking data error detector for the systolic sorting array is shown in Fig. 2.10. Check bits from the *check symbol generator* (*CSG*) are generated based on the coding technique used. They are attached to the corresponding data (information) and propagated through the array but are not processed by the systolic sorting array before arriving the *two-rail checker* (*TRC*) (*TRC* is a two-level *AND-OR* circuit as in [46] and will be described latter). At outputs, these input check symbols are compared with the outputs from the *complement check symbol generator* (\overline{CSG} is a combination of *CSG* which generates check bits for the received data and an inverter at the output of each check bit) through a tree of two-rail checkers. As discussed in the previous section that any input data should not be modified by the systolic sorting array, so the check symbols generated by the \overline{CSG} should be complementary to the check symbols generated by the *CSG* if both the checker and the sorting array are fault-free.

In the case that only one check bit is generated for each codeword (for example, by using the single parity code to detect data error), since N inputs will be processed in parallel, N 1-out-of-2 code outputs ((01) or (10) for code word outputs and (00) or (11) for noncode word

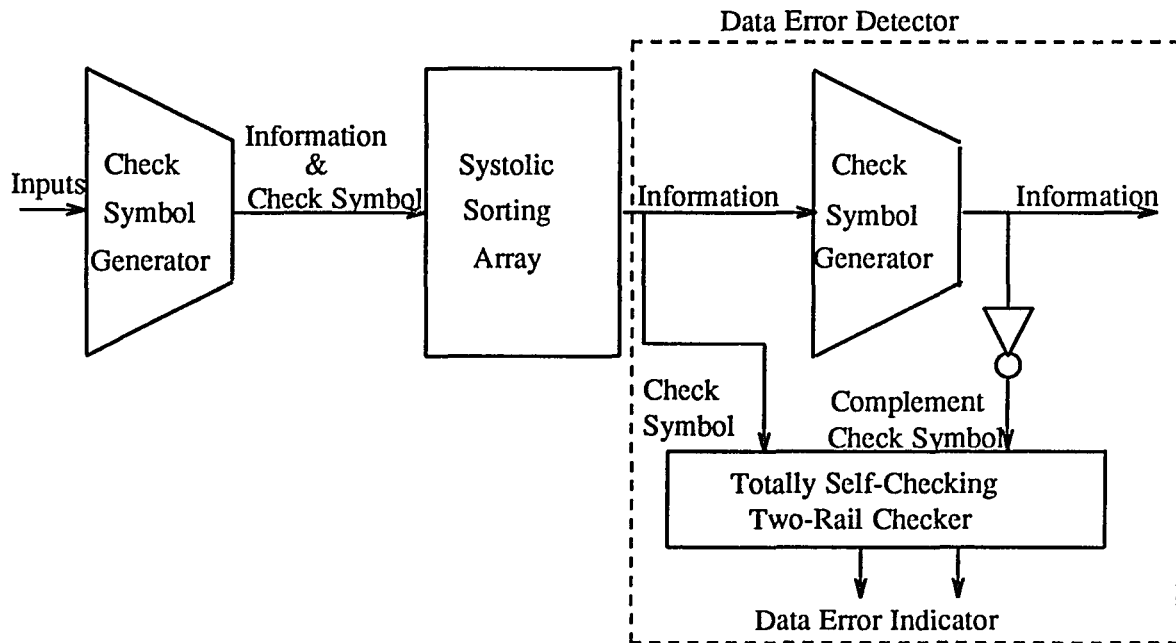


Figure 2.10. Totally self-checking data error detector.

outputs) will be generated in parallel during normal operations. Therefore, a tree of two-rail checkers which maps N input pairs into one output pair can be used to combine these information together and generate a single output (10) or (01) in the normal operation and (00) or (11) as an error message. In order to have a high fault coverage, usually more than one check bit of each data will be generated by the CSG and \overline{CSG} (for example, by using either the Berger code, the modified berger code, or the low-cost code) and therefore, an intermediate-level two-rail checker is required for each code word to map the outputs of check symbols and complement check symbols into a single output pair. Sometimes, the combination of inverters of the \overline{CSG} and the intermediate-level two-rail checkers are called an *equality checker* [46] because it can check whether the input check symbols are the same as the output check symbols or not.

Design of a *TSC* checker for single bit parity code is quite simple. Let $x = (x_n, \dots, x_1)$ be the input data and the code word output from the odd parity generator is (x_n, \dots, x_1, x_0) . Divide the set of variables into two groups, $(x_n, x_{n-2}, \dots, x_1)$ and $(x_{n-1}, x_{n-3}, \dots, x_0)$, and connect variables of each group to the inputs of the parity checker. During the normal operation, the number of 0's in the former group is odd and that in the latter is even, or vice versa. Therefore, for a fault-free output data the outputs of the two parity checker will be either (10) or (01) but never (00) and (11). Verification of the self-checking properties for this checker was given in [49].

Designs of a totally self-checking checker for the modified Berger code and a self-checking checker for the Berger code were presented in [48] and [50] respectively. To avoid the problem of two legal representations of zero during the calculation of residues, either special definitions are required for the modulo 2^m-1 adder in the check symbol generator [51] (where 2^m-1 is the check base of the residue code) or a code translator is added between the equality checkers and the two-rail checkers [52] to design an efficient *TSC* checker for the low-cost code.

It has been proved that the two-rail checker is a totally self-checking checker [46]. The combination of the *CSG* and the \overline{CSG} can be a totally self-checking checker for different coding techniques such as for the simple parity code, [49] the modified Berger code, [48] the Berger code, [50] and the low-cost code [51,52]. Since the output pair from the *CSG* and \overline{CSG} can generate all 0, 1 sequences needed to test the two-rail checker tree, the combination of these two circuits preserves properties of *TSC* [46].

B. Design of a Totally Self-Checking Order Checker

To design a *TSC CS* element, the concept of duplication with comparison is used to generate m -variable ($m = \frac{N-1}{2}$) two-rail code (or 1-out-of-2 code). Every Boolean function $f(x)$ has a corresponding dual function $f_d(x)$ such that $f_d(\bar{x}) = \bar{f}(x)$. If we apply x to the function f and \bar{x} to the function f_d , the resulting output should be complementary to each other and can be used as inputs to a *TSC* two-rail checker. The dual of a Boolean function is found by replacing *AND* operations with *OR* operations, *OR* operations with *AND* operations, 1's with 0's and 0's with 1's [53]. As described in section 2.2, all the cell elements are simple combinational circuits. Hence it is possible to duplicate all the cells in the last two stages with complementary circuitry. This can be further simplified since outputs x_i and y_i are checked by the data error detector. Therefore, only the output information s_i and r_i which indicate whether the cell c_i performs swap or not should be duplicated in order to design the *TSC CS* elements. These *CS* elements which are implemented according to the above method of designing *TSC* circuit will generate paired *swap-indicators* in the form of the 1-out-of-2 code. That is, if a *CS* element has a functional error, its output pair (s_1, \bar{s}_1) will be either (00) or (11) and will be (01) or (10) if it is fault-free.

The stage $N+2$ which is used to correct a functional error should be designed as *TSC* checkers as proved in Theorem 2.3. All output pairs of (s_1, \bar{s}_1) from word-level *TSC CS* elements in this stage will be either (01) if there is no swap operation or (10) if there is any swap operation performed during normal operations and (00) or (11) if there is an error in a *CS* element. Since these 0,1 sequences can completely test the two-rail checker (*TRC*) tree which are used to map N output pairs to form a single output pair, the combination of *TSC CS* elements with *TSC* two-rail checker constitutes a *TSC* checker. The output pair from the two-rail checker

indicates whether there are functional errors (output pair is (11) or (00)) in this stage or not (output pair is (01) or (10)).

In addition to stage $N+2$, CS elements in stage $N+3$ are also designed as TSC circuits to generate m -variable two-rail code such that if there is no functional error in this stage, then the paired output (s_1, \bar{s}_1) of each CS element is either (01) or (10). In addition, if the input sequence to this stage has been ordered correctly, then the *swap-indicators* of all CS elements in this stage should be all 0's and their complement signals are then all 1's, i.e., the paired output (s_1, \bar{s}_1) for all CS elements are (01).

During normal operation, the input sequence to stage $N+3$ will be in correct order if there is no functional error. Therefore, the inputs to the $AND-OR$ pair which is used to map m -variable two-rail code to a single output pair as an error indicator will be all 0's for the OR gate and all 1's for the AND gate (these two gates can be viewed as a tree of two input gates if $m > 2$). The output pair (S, \bar{S}) from the $AND-OR$ circuit should then be (10). This $AND-OR$ circuit can be shown to be code disjoint (this can be proved easily by expanding the truth table to include all possible inputs) and fault secure. The reason that it is fault secure is described in the following. Suppose that a fault has occurred in the OR gate (or AND gate). Depending on the input, a single fault in it may not produce an error or produce an error value which is the 1's complement of the correct value. In the first case, the fault will not affect the output of the gate. In the second case, a fault in the OR gate will not affect the output from the AND gate and a codeword will not be produced. Therefore, for single faults the output of the $AND-OR$ pair is either the correct output or a none codeword and consequently, it is fault secure.

It is impossible for this paired $AND-OR$ circuit to be self-testing under the condition that there is only one code input during the normal operation. Therefore, the NOC which includes

both the *TSC CS* elements and the *AND-OR* circuit will not be a totally self-checking checker because the *swap-indicators* and their complements from *CS* elements in stage $N+3$ can not generate all the input sets required to test the *AND-OR* circuit during the normal operation. But it does have the properties of fault secure and code disjoint which will increase the system reliability.

A complete word-level structure of the fault-tolerant sorting array for $N=5$ is presented in Fig. 2.11. Input data can be encoded with either a parity code, a Berger code, a modified Berger code, or a low-cost code by the check symbol generator (*CSG*) before entering the sorting array. The output sequence is then checked by the *TSC* checkers which include a *DED* to detect data errors in the output sequence and an *NOC* to check whether the output sequence is in nonincreasing order. Stage $N+2$ is implemented as totally self-checking circuits in order to check whether all the compare-and-swap functions performed by the *CS* elements in this stage are correct. The swap error signals from stage $N+2$ will generate an output pair as 11 or 00 if there is an error swapping in this stage.

2.6. Evaluation and Discussion

In this section, the impact of the proposed fault tolerance techniques on fault coverage, area and time overhead will be evaluated. Multiple functional errors can be detected by the *NOC* and any single functional error is masked by the first two additional stages as proved in Theorem 2.1 and Theorem 2.2, respectively. Faults in the *NOC* will be either masked or detected by the *NOC* itself due to its fault secure and code disjoint properties. Coverage of data errors in the proposed sorting array will depend on the complexity of the specific coding technique selected to detect data errors. As mentioned earlier that there is no arithmetic operation

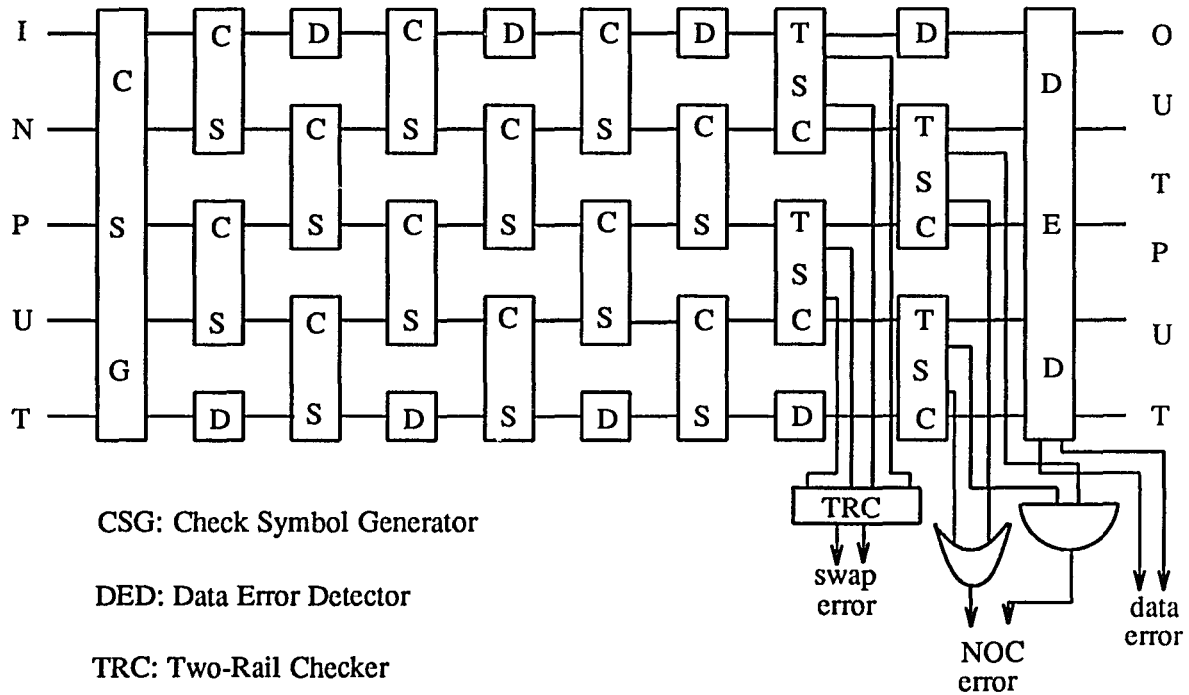


Figure 2.11. A complete fault-tolerant sorting array with $N+3$ stages.

involved in the sorting array and it was observed that some physical defects in the VLSI circuits tend to generate unidirectional errors. Therefore, only the simple parity check code, the Berger code, and the modified Berger code will be considered as potential coding techniques for data error detection. By using the simple parity code, only single bit error in each data word will be detected, however, it incurs the least hardware overhead. All unidirectional errors can be detected by the Berger code but it requires at least 22% overhead than the modified Berger code which has a 93% or more fault coverage of unidirectional errors [48]. Error detection for other types of errors can be achieved by using more complicated codes such as the *AN* code, the check sum code, and the low-cost code. Although they may have higher fault coverage and lower fault masking effect, the requirement of n -bit multipliers, adders, or dividers makes them inefficient for VLSI implementation. For example, with the same number of check bits

generated, the number of full adders required by the low-cost code which can detect undirectional multiple errors as well as errors produced by arithmetic processors is almost twice as it required by the modified Berger code [48].

In the following hardware overhead analysis, the calculation of overhead ratio will be on the gate level. Since the comparison of overhead among the *TSC* Berger checker, the modified Berger checker, and the *TSC* low-cost code checker has been discussed in [48], we will only calculate the overhead ratio for applying the parity code and the modified Berger code. The number of check bits in the modified Berger code in this analysis is assumed to be 2. The number of gates in a 1-bit full adder and a half adder in the check symbol generator of the modified Berger (*MB*) code is 5 and 2, respectively, by assuming that the *EXOR* operation in the adder is performed by an *EXOR* gate.

Let

N = # of input words to be sorted at a time

n = # of bits in each word

g_c = # of gates in each sorting cell = 14

g_t = # of gates in each pair of two-rail checker = 6 (see) [46]

g_i = # of extra gates required for a compare-swap cell to be *TSC* = 6

g_d = # of gates in each pair of *AND-OR* gates = 2

g_p = # of gates in an n -bit parity checker = $(n-1)$ two-inputs *EXOR* gates

g_{mb} = # of gates in an n -bit *MB* checker = $(n-1)/2$ full-adder

+ 2 half-adder + $(n+1)/4$ three-inputs *EXOR* gates = $(11n + 7)/4$.

Then

A = # of gates in the original sorting array = $g_c N(N-1)n/2$

B = # of gates in the three additional stages = $g_c 3(N-1)n/2$

$C = \# \text{ of extra gates for stages } N+2 \text{ and } N+3 \text{ to be } TSC = 2g_i(N-1)n/2$

$D = \# \text{ of gates which map } (N-1)/2 \text{ output pairs of } (s_1, \bar{s}_1) \text{ to one pair in stage } N+2 = g_i[(N-1)/2 - 1]$

$E = \# \text{ of gates to put } s_1 \text{ and } \bar{s}_1 \text{ of a } CS \text{ element in stage } N+3 \text{ together} = g_d[(N-1)/2 - 1]$

$F = \# \text{ of gates in a parity checker} = 2g_pN$

$G = \# \text{ of gates which map } N \text{ to one output pair in the parity checker} = g_i(N-1)$

$H = \# \text{ of gates in the } MB \text{ checker} = 2g_{mb}N$

$I = \# \text{ of gates which map } N \text{ to one output pair in the } MB \text{ checker} = g_i(N-1).$

Therefore, the respective overhead ratio for using the *TSC NOC* with either the *TSC* parity checker (r_p) or the *TSC MB* checker (r_{mb}) is listed below :

$$\begin{aligned}
 r_p &= \frac{(B+C+D+E+F+G)}{A} 100\% = \frac{P}{A} 100\% \\
 &= \frac{42(N-1)n+12(N-1)n+6(N-3)+2(N-3)+4N(n-1)+12(N-1)}{14N(N-1)n} 100\% \\
 &= \frac{54(N-1)n+8(N-3)+4N(n-1)+12(N-1)}{14N(N-1)n} 100\% \\
 &\approx \frac{58}{14N} 100\% \quad (N \gg 1)
 \end{aligned}$$

$$\begin{aligned}
 r_{mb} &= \frac{(B+C+D+E+H+I)}{A} 100\% = \frac{MB}{A} 100\% \\
 &= \frac{54(N-1)n+8(N-3)+N(11n+7)+12(N-1)}{14N(N-1)n} 100\% \\
 &\approx \frac{65}{14N} 100\% \quad (N \gg 1)
 \end{aligned}$$

From the above analysis, we can see that the cost B , C , D and E , are required to implement a *TSC NOC*. Therefore, a general overhead ratio for the proposed fault tolerance tech-

niques can be written as :

$$\begin{aligned}
 r &= \frac{(B+C+D+E+T)}{A} 100\% \\
 &= \frac{54(N-1)n+8(N-3)+T}{14N(N-1)n} 100\% \\
 &\approx \frac{54+c}{14N} 100\% \quad (N \gg 1)
 \end{aligned}$$

where the values of T and c will depend on the complexity of the selected coding technique.

Examples of overhead ratios on different values of n and N by using simple parity code are shown in Table 2.2. From the table, it is observed that the difference between overhead ratios for arrays with 8-bit input words and 16-bit input words is very small since n does not dominate the equation. The overhead ratio drops in proportion to $\frac{1}{N}$ and therefore, the overhead ratio is smaller for an array with a larger input set.

Table 2.2. Overhead ratios.

n	N	P	A	r
8	25	11532	67200	17.16
8	49	23052	263424	8.75
8	81	38412	725760	5.29
8	121	57612	1626240	3.54
8	169	80652	3179904	2.54
8	225	107532	5644800	1.90
16	25	22700	134400	16.89
16	49	45356	526849	8.61
16	81	75564	1451520	5.21
16	121	113324	3252480	3.48
16	169	158636	6359808	2.49
16	225	211500	11289600	1.87

The proposed fault-tolerant sorting array is highly pipelined. Once the pipe is filled we can get an output for every clock cycle although the time latency required to fill the pipe was increased by 3 clocks due to the three additional stages for error detection and correction. The analysis shows that the hardware overhead is less than 10% if $N > 42$ and the time overhead approaches zero after the pipe is filled.

If we apply the *RESO* [30] approach which detects errors by comparing the unshifted output sequence with the shifted output sequence, it requires 100% time overhead and one extra cell in each *CS* element to sort the shifted input sequence in order to detect any single data error. Even if we do not include the shifting circuits and comparators for comparison, the hardware overhead ratio is

$$r_{reso} = \frac{14N(N-1)}{14N(N-1)n} 100\% = \frac{1}{n} 100\%.$$

The AT^2 ratio of *RESO* over the proposed method is equal to

$$R_{AT^2} = \frac{\frac{1}{n}}{\frac{54+c}{14N}} \frac{4}{1} = \frac{56N}{(54+c)n}$$

For $N \gg n$, $R_{AT^2} \gg 1$ which shows that the area-time cost of *RESO* (with the error detection capability only) is higher than our method.

2.7. Summary

A novel fault tolerance technique was presented for a systolic sorting array based on the odd-even transposition sort algorithm. Functional and data errors are detected by additional stages and a simple coding technique respectively. Based on the discovered properties and the developed fast on-line fault diagnosis procedure, these errors can be corrected either automati-

cally or by bypassing and reconfiguration. Hardware overhead for fault tolerance is about $(54+c)/14N$ and only 3 clocks delay is incurred in the pipeline. Since the sorting array is two-level pipelined and all the checkers are implemented to be fault secure or totally self-checking, it is well applicable to real-time applications which require high throughput as well as high reliability. The error detection techniques in this chapter can be applied to sorting arrays based on other sorting algorithms with either two-level pipelined or bit-level serial structure [54].

CHAPTER 3.

THE DEFECT-TOLERANT WSI SORTING NETWORKS

3.1. Introduction

Recently, the fast growing computer vision, image processing, and digital signal processing techniques [55,56,57,58] enforce the sorters to process even more input data in a shorter period of time for real-time applications. According to Thompson's [4] analysis, only two of the thirteen sorters discussed in his chapter were designed with high degree of concurrency and thus suitable for real-time applications. One uses the odd-even transposition sort [5] which requires $N \cdot (N-1)/2$ basic sorting elements (N is the number of input data to be sorted) to completely sort the input sequence with the concurrency factor N . This sorting algorithm is widely used in VLSI systems [32,59,33,60], because it has the advantages of regular cell structure and simple communication scheme which render it easily implementable and reconfigurable in VLSI technology [61,62]. However, this is a hardware intensive architecture since it requires $O(N^2)$ sorting elements to sort N input data. The other one uses the Batcher's bitonic sort [6] which requires $(N/2)[\log_2 N \cdot (\log_2 N + 1)]/2$ sorting elements with the concurrency factor $[\log_2 N \cdot (\log_2 N + 1)]/2$. Although this architecture has the advantage of logarithmic *area-time*² (AT^2) cost, it is difficult to implement in VLSI if N is very large, due to its complex communication scheme. Even with the modified sorting networks such as the perfect shuffle sort [7] or the balanced sort [63], which are much more regular than the bitonic sort, they are still hard to implement in VLSI due to their complex interconnections. The results in [64] show that the

minimum area required to lay out an m -line perfect shuffle interconnection networks grows as m^2 . This problem is even more significant when the sorter is implemented in WSI (Wafer Scale Integration) which can have a huge number of sorting elements fabricated in a single wafer. In addition, due to the large area and the processing technology limitation, defects seems unavoidable in WSI implementation. Therefore, the networks need to have defect tolerance capabilities. Although various approaches on fault tolerant interconnection networks for shared memory multiprocessors have been proposed [65, 66, 67, 68, 43, 69], these techniques can not be applied to sorting networks since every interconnections in the sorter are active at any given time and the data movements are highly pipelined.

Therefore, in order to obtain a good *area-time* tradeoff, in section 3.2 we present a novel sorting network which is designed to be hierarchical and modular and retains advantages of both sorting networks discussed above. The hierarchical modular sorting network (*HMSN*) is based on the tradeoffs between the simple communication scheme of the odd-even transposition sort and the fast convergent speed of the bitonic sort. In section 3.3, an approach to determine the optimal sorting capability at each level is proposed based on the technology constraints and the requirement of hardware area. A cost function is derived and simulations are performed to find the minimum cost with respect to various parameters.

Although the *HMSN* is highly modular, it is still difficult to exclude faulty elements in the network and replace them by redundant elements since the connections between stages in the bitonic sorter are irregular and complex. Therefore, in section 3.4 networks with regular interconnections are derived and shown to be equivalent to the bitonic network and therefore can replace it. In section 3.5, defect tolerant structures are presented. Spare sorting elements are incorporated in every level of the hierarchy and they not only can replace defective sorting ele-

ments in the corresponding level but also can be used to correct run-time errors. Detailed yield analysis is done in section 3.6 which shows that our approach is indeed very effective in comparison with other structures.

3.2. Hierarchical Modular Sorting Networks

As discussed in section 3.1, the Batcher's bitonic sort has the advantage of having a logarithmic *area-time*² (AT^2) cost over other sorters. However, if N is very large, it is difficult to implement the N -input bitonic sorter in a single chip VLSI [64] or WSI [70] due to the complex and long interconnections. As shown in the middle of Fig. 3.1, both shuffle and butterfly interconnections are used in the bitonic sorter and the longest interconnection exists between sorting elements which are $n/2$ elements away from each other if there are n elements in each stage. Although the odd-even transposition sorter is a hardware intensive architecture (it requires $O(N^2)$ sorting elements to sort N inputs), it has the advantage of having simpler and shorter interconnections. As shown in the left of Fig. 3.1, every sorting element only communicates with its two nearest neighbors and hence the odd-even sorter is more suitable for implementation in VLSI and WSI. Therefore, in order to have advantages of fewer processing elements (area cost) as well as less wire complexity and faster convergence in sorting, the sorting network can be decomposed into a two-level structure with the bitonic sorter in the bottom level and the odd-even transposition sorter in the top level. For example, if $N = b' \times p'$, the network can be decomposed such that the bitonic sorter in the bottom level will sort b' inputs and the top-level odd-even sorter will merge the p' sets of data with b' sorted inputs in each set.

The reason why a two-level sorting network can reduce the wire complexity can be demonstrated with the following example. If a one-level bitonic sorter has $N=1024$ inputs, we

know that 55 stages with 512 sorting elements in each stage are required to complete the sorting process and thus, there are 1024 interconnections between two stages. Since the complexity of shuffle and butterfly interconnections grows with the square of the number of elements to be connected between two neighbor stages, if a bitonic sorter with 1024 shuffle or butterfly interconnections is decomposed into a two-level sorter with $b' \times p' = 256 \times 4$ or 128×16 , the number of sorting elements in each stage will be reduced significantly to 128 or 64, and the number of interconnections is reduced from 1024 to 256 or 128. Therefore, the original wire complexity which is in proportion to 1024^2 will be reduced to 256^2 or 128^2 , and thus simplify the wire complexity considerably.

Although this two-level sorting network now has a good *area-time* cost measure, it is difficult to incorporate redundancy and reconfigure for surviving from defects since the bottom-

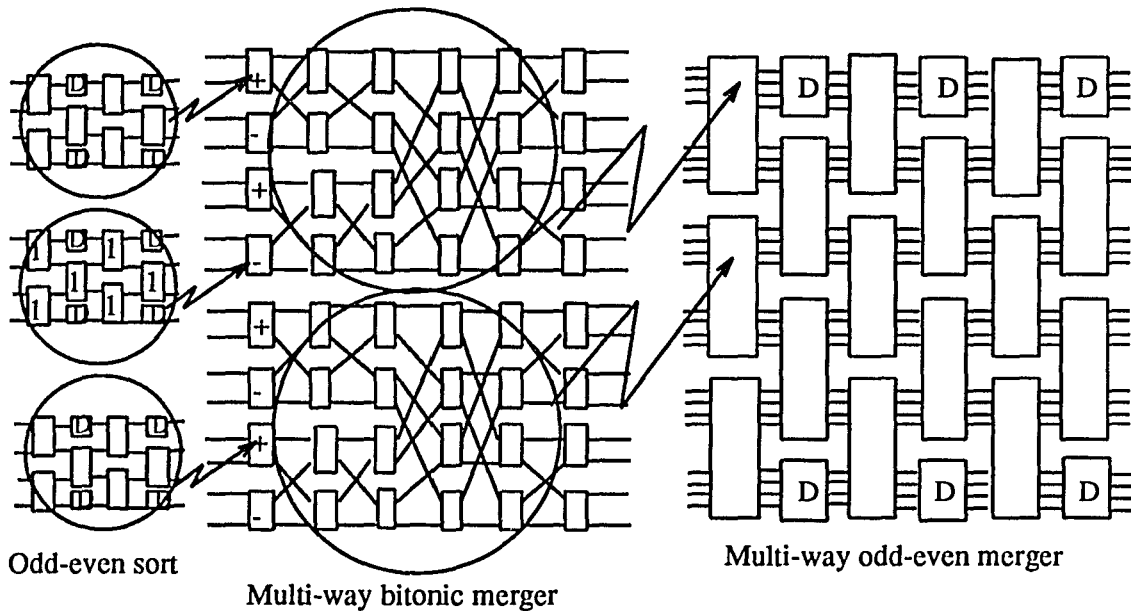


Figure 3.1. Hierarchical sorter.

level bitonic sorter has irregular shuffle and butterfly interconnections. Therefore, it is not cost effective to use this architecture for WSI implementation where reconfiguration is necessary to tolerate defects. To minimize the cost to survive from defects, the easily reconfigurable odd-even transposition sorter can be used as the bottom level sorter to replace a sorting element in each bitonic sorter (Reconfiguration on the odd-even transposition sorter and the bitonic sorter will be discussed in section 3.5).

Therefore, the sorting network has three levels. Let $N = p_1 \times b \times p_2$. Then, each bottom-level odd-even sorter can sort p_1 inputs, each middle-level bitonic sorter can merge b sets of sorted inputs with p_1 inputs per set, and the top-level odd-even sorter can merge p_2 sets of inputs with $b \cdot p_1$ inputs in each set. In the rest of this chapter, a sorting element will be referred as a *cell* at the bottom level, a *submodule* at the middle level, and a *module* at the top level. Each bottom-level odd-even sorter has p_1 stages with $p_1/2$ *cells* in an odd stage and $(p_1/2)-1$ *cells* in an even stage if p_1 is even [5]. If p_1 is odd, there are $(p_1-1)/2$ *cells* in a stage. A data register "*D*" in the odd-even sorter is used as a buffer to synchronize the data movements. We refer a middle-level bitonic sorter in Fig. 3.1 as a *multi-way bitonic merger*. A cell (submodule) marked with a "1" ("-") means that the outputs from it are in monotonic decreasing order, otherwise, the outputs are in monotonic increasing order.

It can be shown by using the method similar to that in [5] for *merge-sort* that the multi-way bitonic merger in the middle level with a total of $(\log_2 b + 1) \cdot (\log_2 b + 2) / 2$ stages can completely sort $p_1 \cdot b$ inputs if there are b (b needs to be a power of two) submodules in each stage and each module can sort p_1 inputs. In the top level, the odd-even sorter is referred as a *multi-way odd-even merger* which can merge p_2 sets of $p_1 \cdot b$ sorted inputs into the correct order. The multi-way odd-even merger has $2p_2 - 1$ stages with p_2 modules in each stage and

can merge p_2 sets of $p_1 \cdot b$ sorted outputs into the correct order if each module can sort $p_1 \cdot b$ inputs. An example three-level sorter is shown in Fig. 3.1 where the four-input odd-even transposition sorter is used in the bottom level. Depending on the number of inputs to be processed, each level can be furthermore decomposed. For example, if p_2 is still very large after the decomposition, then the top level can be further decomposed into two levels with one based on the bitonic sort to save area and the other one based on the odd-even sort.

In addition to the bitonic sorter, the perfect shuffle sorter [5, 7] can also be a good candidate for the middle level. In a perfect shuffle sorter, $\log_2 n$ blocks are configured as an *Omega* interconnection network, *i.e.*, interconnections between blocks are shuffle connections. Each block of the perfect shuffle sorter is also constructed as an *Omega* network except that switching elements in the original *Omega* network are replaced by sorting elements in the perfect shuffle sorter. That is, with a total of $(\log_2 n)^2$ stages and $n/2$ elements in each stage, the perfect shuffle sorter can completely sort n inputs. If we replace each bitonic sorter in Fig. 3.1 by a perfect shuffle sorter, a *multi-way perfect (shuffle) merger* is formed.

Although the multi-way bitonic merger uses less sorting submodules and incurs less time latency ($\log_2 n \cdot (\log_2 n + 1)/2$ stages) to fill the pipeline than the perfect merger ($(\log_2 n)^2$ stages), the multi-way perfect merger has the advantage that it has the same interconnection pattern between stages in a block and between blocks. This repetitive architecture can simplify both the design and the operation complexity compared with the recursive architecture of the bitonic merger. However, both the bitonic merger and the perfect merger need more than one type of submodules which may increase the implementation complexity.

Recently, a new sorting network, the balanced sorter (as shown in Fig. 3.2), was proposed in [63]. Although it requires the same number of blocks as the perfect shuffle sorter to sort n

inputs and each block is also configured as an *Omega* network, permutations between blocks are different. Instead of shuffle connections, τ permutations exist between blocks of the balanced sorter as shown in Fig. 3.2. This balanced sorter is essentially equivalent to the bitonic sorter [71]. If we replace each bitonic sorter in Fig. 3.2 by a balanced sorter, a *multi-way balanced merger* is formed. A multi-way balanced merger has some advantages over a multi-way bitonic merger: (1) unlike the multi-way bitonic merger which does not have uniform sorting submodules, the multi-way balanced merger contains only one type of submodules (2) interconnections between stages in each block are the same and the permutations between blocks are all τ connections. The uniform submodule property in (1) is also an advantage over the perfect shuffle merger.

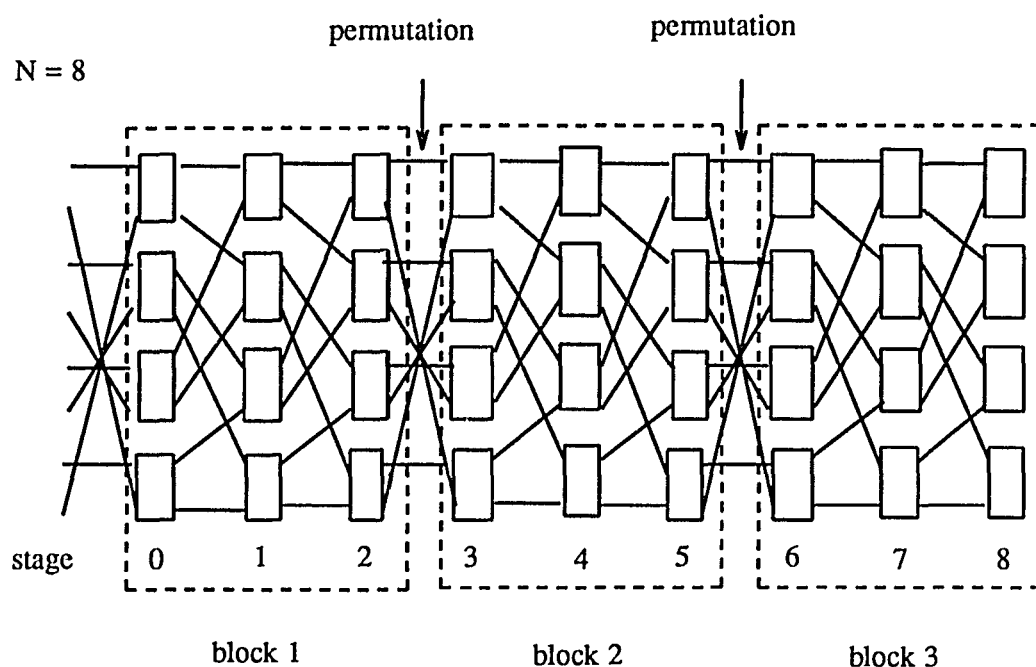


Figure 3.2. The balanced sorter.

Therefore, the multi-way balanced merger is more suitable for WSI implementation because of its uniform submodules and regularly repeated architecture. If N is large and the timing requirement is not very critical, we can even use only one block of the balanced sorter and recirculate the outputs of this block to its inputs for $\log_2 N$ times until the sequence is ordered. Inherent fault tolerance properties of the balanced sorting network were discussed in [26,27]. By recirculating all output lines to the corresponding input lines or duplicating the last block, a functional fault which generates an incorrect swap will be recovered automatically.

3.3. Optimal Decomposition

We know that the *HMSN* can have multiple levels for large N with odd-even transposition sorters at both the bottom and the top level. In this section, we will present the analysis procedures for choosing an optimal sorting capability (number of inputs or sets of inputs) of each level based on the wire complexity and the hardware cost. We will assume that the multi-way balanced mergers are used at the intermediate levels in the analysis, however, similar analysis can be performed if the multi-way bitonic mergers are used.

For a three-level sorter, let $N = p_1 \times b \times p_2$ and therefore, the total number of cells is

$$N_h = p_1(p_1 - 1) \times b(\log_2 b + 1)^2 \times p_2(2p_2 - 1)/2. \quad (3.1)$$

It should be noted that p_1 in equation (1) should be greater than 2 to form a sorter in the bottom-level. Otherwise, the network will be a two-level structure since a submodule of the balanced sorter can sort two inputs directly [10,72]. The ratio $r_{h/o}$ of N_h over the number of sorting elements in a single-level odd-even sorter which is $N(N-1)/2 = p_1 b p_2 (p_1 b p_2 - 1)/2$ is then

$$r_{h/o} = \frac{p_1(p_1-1) \times b(\log_2 b + 1)^2 \times p_2(2p_2-1)}{p_1 b p_2 (p_1 b p_2 - 1)} = \frac{(p_1-1) \times (\log_2 b + 1)^2 \times (2p_2-1)}{p_1 b p_2 - 1} < \frac{2(\log_2 b + 1)^2}{b} \quad (3.2)$$

$r_{h/o} \approx 1$ when $b=128$. Thus, the *HMSN* will have less sorting elements than the single-level odd-even sorter if $b > 128$. From equation (2), we can see that if $b \gg (\log_2 b + 1)^2$, we will have $r_{h/o} \ll 1$. Therefore, b should be as large as possible under the constraints imposed by the technology and wire complexity in order to reduce the number of sorting cells. Thus, the optimal b is technology dependent. In the following analysis we will assume that b is known to be equal to an optimal value b_{\max} which depends on the technology and the wire complexity.

After the value of b has been determined in a three-level *HMSN*, we can find the values for p_1 and p_2 . From equation (1), since both N and b are fixed, $\log_2 b$ as well as $p_1 \times p_2$ (let it be represented as p) are fixed, the minimization of N_h is then equivalent to minimizing $(p_1-1)(2p_2-1) = 2p_1 p_2 - 2p_2 - p_1 + 1$. By maximizing $2p_2 + p_1$ under the constraint that $p_1 \times p_2$ is equal to a constant p , we will have a minimum N_h . This means that p_1 , which is an integer factor of p , should be as small as possible but greater than 2.

However, with redundancies included in every level, finding the minimum N_h with respect to p_1 is very difficult. Simulation is necessary to select the minimum N_h and thus determine p_1 and p_2 . The reason is that the minimization procedure involves finding the minimum value of a fourth order function of p_1 and p_1 not only has to be discrete (p_1 is an integer) but also must be a factor of p . Let N_{h-r} be the number of sorting elements in a *HMSN* with redundancy and assume that there are d , l , and n redundant rows as well as f , m , q redundant columns in each sorter at the bottom-level, middle-level, and top-level, respectively. Then,

$$N_{h-r} = \left(\frac{p_1-1}{2} + d \right) \cdot (p_1 + f) \cdot (b + l) \cdot [(\log_2 b + 1)^2 + m] \cdot (p_2 + n) \cdot (2p_2 - 1 + q). \quad (3.3)$$

Finding the minimum N_{h-r} with respect to various p_1 is equivalent to finding the minimum C where

$$\begin{aligned}
C &= (p_1 - 1 + 2d) \cdot (p_1 + f) \cdot (p_2 + n) \cdot (2p_2 - 1 + q) \\
&= [p_1 p_2 + (2d - 1)p_2 + n p_1 + n(2d - 1)] \cdot [2p_1 p_2 + 2f p_2 + (q - 1)p_1 + f(q - 1)].
\end{aligned} \tag{3.4}$$

Since b in equation (3) and the redundancies added to each sorter are fixed, only p_2 ($= p/p_1$) is related to p_1 in finding the minimum N_{h-r} with respect to p_1 .

Example cases with various amounts of redundancy are used to illustrate how to find the minimum C (or N_{h-r}) with respect to a given p . It should be noted that N_{h-r} is equal to $k \times C$ where k can be viewed as a constant and is equal to $(b+l) \cdot [(\log_2 b + 1)^2 + m]/2$. These example cases have one or two redundant rows in a cell, submodule, or module. Redundant columns can also be included in a cell or module since they, as will be discussed in section 3.5, can simplify the system on-line reconfiguration process. The amount of redundancy in a level for each case is shown in Table 3.1 where nr represents n redundant row and mc represents m redundant columns. Tables 3.2-3.5 present the results for these example cases with various p values (*i.e.*, different array sizes). The C_{\min} is the minimum C and the corresponding p_1 is listed as $p_{1\min}$. For case 1, one redundant row is incorporated in a sorter at every level, then

$$N_{h-r} = (p_1 + 1) \cdot p_1 \cdot (b + 1) \cdot (\log_2 b + 1)^2 \cdot (p_2 + 1) \cdot (2p_2 - 1) / 2,$$

$$C = (p_1 + 1) \cdot p_1 \cdot (p_2 + 1) \cdot (2p_2 - 1) = (p_1 p_2 + p_1) \cdot (2p_1 p_2 + 2p_2 - p_1 - 1).$$

The global minimum C will occur at the value of p_1 which satisfies the equation $p + 2p_2 - 2p_1 = 1$

Table 3.1. The amount of redundancy for each case.

	Case No.							
level	1	2	3	4	5	6	7	8
bottom	1r	1r2c	1r	1r2c	2r	2r2c	2r	2r2c
middle	1r	1r	1r	1r	2r	2r	2r	2r
top	1r	1r	1r2c	1r2c	2r	2r	2r2c	2r2c

($p=p_1 \times p_2$). However, since both p_1 and p_2 are discrete, simulation will be necessary to determine the values of p_1 and p_2 which minimize N_{h-r} .

In Fig. 3.3 we also show the cost versus p_1 graphically for cases 1, 2, 3 and 4 with $p=100$ and $p=105$. The cost decreases rapidly before $p_1=p_{1min}$ for case 2 which has two redundant columns in the bottom-level sorter and increases rapidly after $p_1>p_{1min}$ for case 3 which has two redundant columns in the top-level sorter. Since case 4 has two redundant columns in both the bottom-level and the top-level sorters, the curve in Fig. 3.3 shows the cost decreases before

Table 3.2. The cost C with $p=20$.

	<i>Case No.</i>							
p_1	1	2	3	4	5	6	7	8
4	1080	1620	1320	1980	1764	2646	2156	3234
5	1050	1470	1350	1890	1680	2352	2160	3024
10	990	1188	1650	1980	1560	1872	2600	3120
C_{min}	990	1188	1320	1890	1560	1872	2156	3024
p_{1min}	10	10	4	5	10	10	4	5

Table 3.3. The cost C with $p=100$.

	<i>Case No.</i>							
p_1	1	2	3	4	5	6	7	8
4	25480	38220	26520	39780	37044	55566	38556	57834
5	24570	34398	25830	36162	34320	48048	36080	50512
10	22990	27558	25410	30492	29640	35568	32760	39312
20	22680	24948	27720	30492	28980	31878	35420	38962
25	22750	24570	29250	31590	29400	31752	37800	40824
50	22950	23868	38250	39780	31800	33072	53000	55120
C_{min}	22680	23868	25410	30492	28980	31752	32760	38962
p_{1min}	20	50	10	10/20	20	25	10	20

Table 3.4. The cost C with $p=105$.

	Case No.							
p_1	1	2	3	4	5	6	7	8
3	29808	49680	30672	51120	45954	76590	47286	78810
5	27060	37884	28380	39732	37720	52808	39560	55384
7	25984	33408	27776	35712	34510	44370	36890	47430
15	24960	28288	28800	32640	31590	35802	36450	41310
21	24948	27324	30492	33396	31752	34776	38808	42504
35	25200	26640	35280	37296	33250	35150	46550	49210
C_{\min}	24948	26640	27776	32640	31590	34776	36450	41310
$p_{1\min}$	21	35	7	15	15	21	15	15

Table 3.5. The cost C with $p=200$.

	Case No.							
p_1	1	2	3	4	5	6	7	8
4	100980	151470	103020	154530	144144	216216	147056	220584
5	97170	136038	99630	139482	132720	185808	136080	190512
8	91728	114660	95472	119340	116424	145530	121176	151470
10	90090	108108	94710	113652	111540	133848	117260	140712
20	87780	96558	97020	106722	104880	115368	115920	127512
25	87750	94770	99450	107406	105000	113400	119000	128520
40	88560	92988	108240	113652	108360	113778	132440	139062
50	89250	92820	114750	119340	111300	115752	143100	148824
100	90900	92718	151500	154530	123600	126072	206000	210120
C_{\min}	87750	92718	94710	106722	104880	113400	115920	127512
$p_{1\min}$	25	100	10	20	20	25	20	20

$p_{1\min}$ and then increases after $p_1 = p_{1\min}$. The cost does not change significantly with respect to p_1 for case 1 with no redundant columns.

Comparing case 2 with case 3 and case 6 with case 7 in Tables 3.2-3.5, we see that the minimum cost of adding two redundant columns in the bottom-level sorter is less than that of adding two redundant columns in the top-level sorter. Case 4 has the highest hardware cost

among the first four cases and it costs about 20-25% more than case 1 or 2. Case 8 has the highest cost among all cases since it has the most redundancy in every level. However, the optimal amount of redundancy depends not only on the area overhead but also on the yield improvement achieved over the original structure with no redundancy. Therefore, the results on these cases will be used in section 3.6 to determine the optimal amount of redundancy in each level.

We also discussed in section 3.2 that depending on the number of inputs, the bottom or the top-level sorter can be further decomposed into a sorter with two or more levels. The bottom(top) level sorter can be decomposed with the bitonic mergers or balanced mergers in the higher (lower) levels and a odd-even merger in the lowest (highest) level. Since p_1 and p_2 can be determined from the simulation, in the following analysis we will show that when the bottom or top level sorter should be further decomposed.

There are $p_1(p_1-1)/2$ cells and $p_2(2p_2-1)$ modules in a bottom-level sorter and a top-level sorter, respectively. Let the bottom-level sorter and the top-level sorter be further decomposed such that $p_1=p_1' \cdot b_1'$ and $p_2=p_2' \cdot b_2'$. Compare the numbers of sorting cells and modules in these two levels before and after decomposition and let the ratios be r_1 and r_2 , respectively, we have

$$r_1 = \frac{p_1(p_1-1)}{p_1' b_1' (p_1'-1) \log_2 b_1' + 1 \log_2 b_1' + 1}, \quad (3.5)$$

$$r_2 = \frac{p_2(2p_2-1)}{p_2' b_2' (2p_2'-1) \log_2 b_2' + 1 \log_2 b_2' + 1}. \quad (3.6)$$

For $p_1=64$ and $p_1'=4$ (since p_1' should be greater than 2 and b_1' should be as large as possible), we have $r_1=63/75$ and if $p_1=128$ we will have $r_1=127/108$. This means that if p_1 is further decomposed, the network will have more cells than before decomposition if $p_1 \leq 64$.

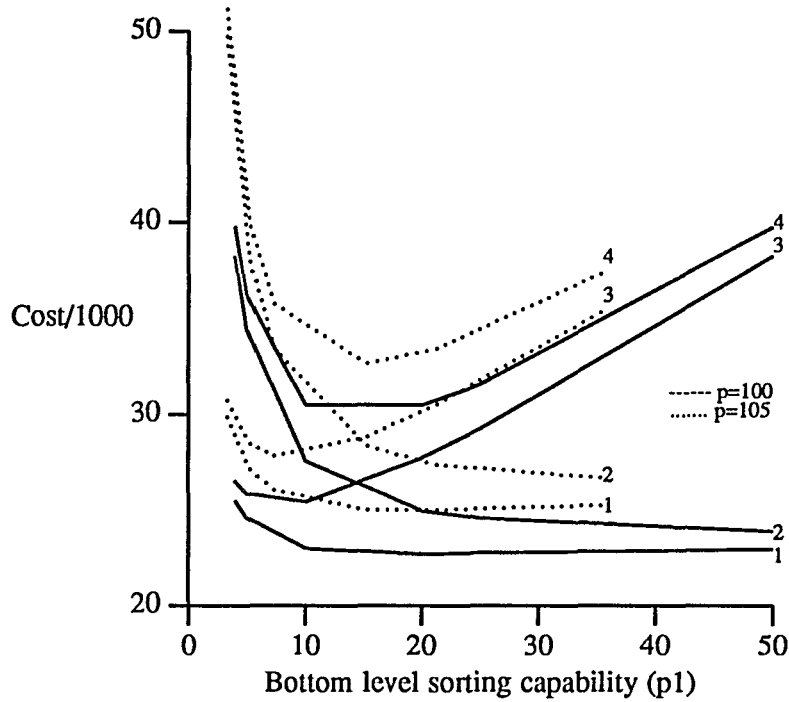


Figure 3.3. Cost C vs. p_1 for cases 1, 2, 3 and 4 with $p = 100$ and 105.

Since b_1' should be a power of 2, we know that further decomposition is profitable in the bottom-level if p_1 is greater than 128. Similarly, we can derive that p_2 should be greater than 256 if the top-level sorter is to be decomposed. From Tables 3.2-3.5, we see that in our example cases every p_1 or p_2 are less than 128 or 256, respectively. Therefore, normally a three-level network is sufficient for most applications unless a huge number of inputs (more than $128 \times 256 \times 256$ inputs, *i.e.*, $p = 128 \times 256$ and $b_{\max} = 256$) is to be sorted. Practically, it may not be possible to implement a sorting network to sort more than $128 \times 256 \times 256$ inputs in a single wafer. Hence, in the rest of this chapter we will concentrate on the three-level structure only.

3.4. Easily Reconfigurable Equivalent Networks

In the middle-level of the *HMSN*, both the multi-way bitonic merger and the multi-way balanced merger are considered in this chapter. The balanced merger has the unique properties of uniform cell structure and regularly repeated architecture. Advantages of the bitonic merger are that it has the fewest submodules and the least pipeline latency among all real-time sorting networks, and the number of submodules is approximately half of that in other mergers. Although the bitonic merger is not a regular and repetitive architecture originally, after an equivalent network transformation described in this section, the resulting network will have a regular structure and simpler interconnections.

However, there still exist shuffle type interconnections in these two mergers which are very difficult to reconfigure to exclude faulty elements and some modifications are necessary to make these networks easily reconfigurable. It has been shown in [73] that the modified data manipulator (see Fig. 3.4(b)) is topologically equivalent to the Omega network (see Fig. 3.4(a)). In addition to the topological equivalence, these two networks are functionally equivalent [70] such that without any modification the shuffle connected *Omega* network in a sorting block of the balanced merger can be replaced by the modified data manipulator (detailed proofs will be discussed in appendix section A.1). Since the modified data manipulator has simpler interconnections, the resulting network is easier to reconfigure (will be discussed in section 3.5) around faulty submodules.

The shuffle connections in the multi-way balanced merger can now be simplified by replacing the *Omega* network in each sorting block with the modified data manipulator. The remaining shuffle permutations (σ) in the multi-way bitonic merger and τ permutations in the multi-way balanced merger will be defined here and then replaced by the equivalent switching

N = 16

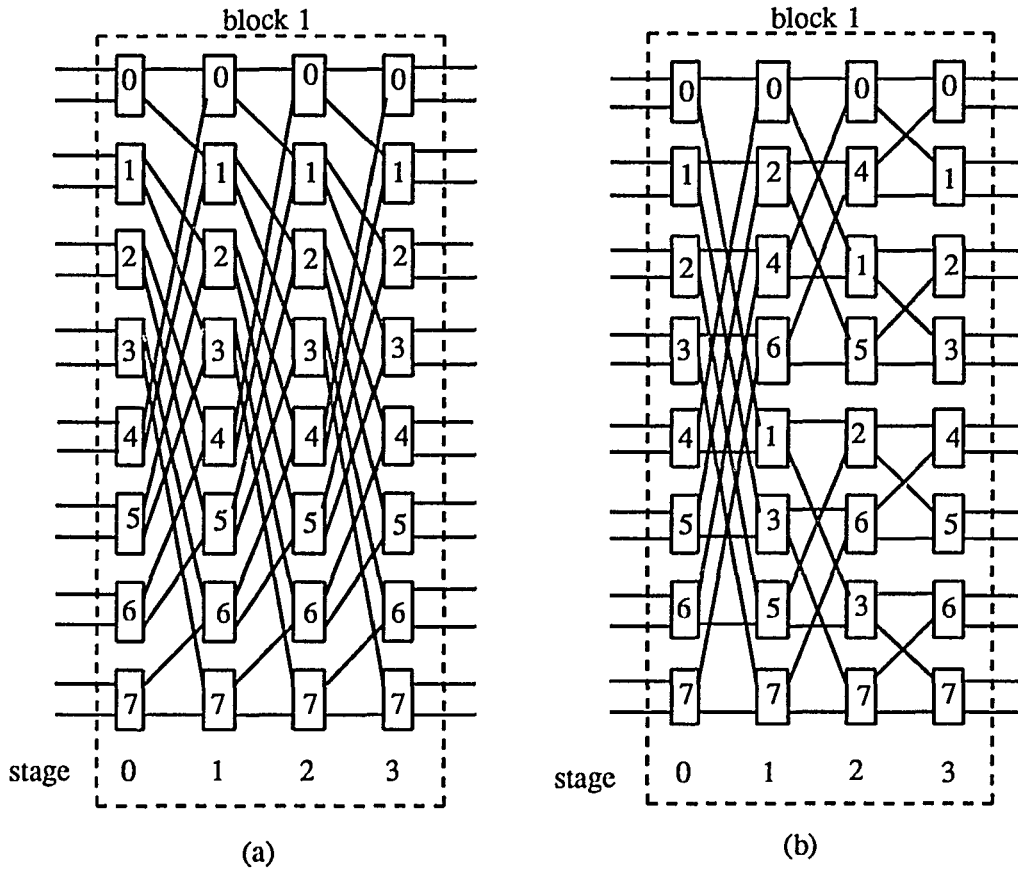


Figure 3.4. (a) Omega network and (b) modified data manipulator.

network so that these two multi-way mergers can be replaced by a more regular and reconfigurable equivalent multi-way merger (detailed proofs will be discussed in appendix section A.2).

Let a sequence $A = \{0, 1, \dots, 2^n - 1\}$ ($N = 2^n$) be represented by $p_{n-1} \dots p_0$, and let τ and σ be two permutations of A . The shuffle permutation σ is defined as $\sigma : A \rightarrow A$ with $\sigma(p_{n-1} \dots p_0) = p_{n-2} p_{n-3} \dots p_1 p_0 p_{n-1}$. An example of σ permutation with $N = 16$ is shown in Fig. 3.5(a). The permutation τ is defined as $\tau : A \rightarrow A$ with $\tau(p_{n-1} \dots p_0) = p_{n-1} \dots p_0$ if $p_0 = 0$ and $\tau(p_{n-1} \dots p_0) = \bar{p}_{n-1} \bar{p}_{n-2} \dots \bar{p}_1 p_0$ if $p_0 = 1$. An example of τ permutation with $N = 16$ is

shown in Fig. 3.6(a). The Banyan permutation r is formed by setting all switching elements in the Banyan's interconnection network (this network can be viewed as a reverse network of the modified data manipulator in network topology) in straight connection states (see Fig. 3.5(b)). The ψ permutation is formed by setting every switching element in the modified data manipulator either in straight or in exchange state. The switching elements in stage i ($i=0$ to $n-1$) with positions represented by $p_{n-1} \dots p_1$, will be in exchange state ($p_{n-1} \dots p_1 p_0 = p_{n-1} \dots p_1 \bar{p}_0$) if $p_{n-i}p_{n-i-1}=01$ or 10 , or in straight state ($p_{n-1} \dots p_1 p_0 = p_{n-1} \dots p_1 p_0$) if $p_{n-i}p_{n-i-1}=00$ or 11 . An example of ψ permutation with $N = 16$ is shown in Fig. 3.6(b). It has been shown [70] that : (1) the shuffle permutation σ is topologically equivalent to the Banyan permutation r , (2) The τ permutation (Fig. 3.6(a)) is topologically equivalent to ψ permutation (Fig. 3.6(b)).

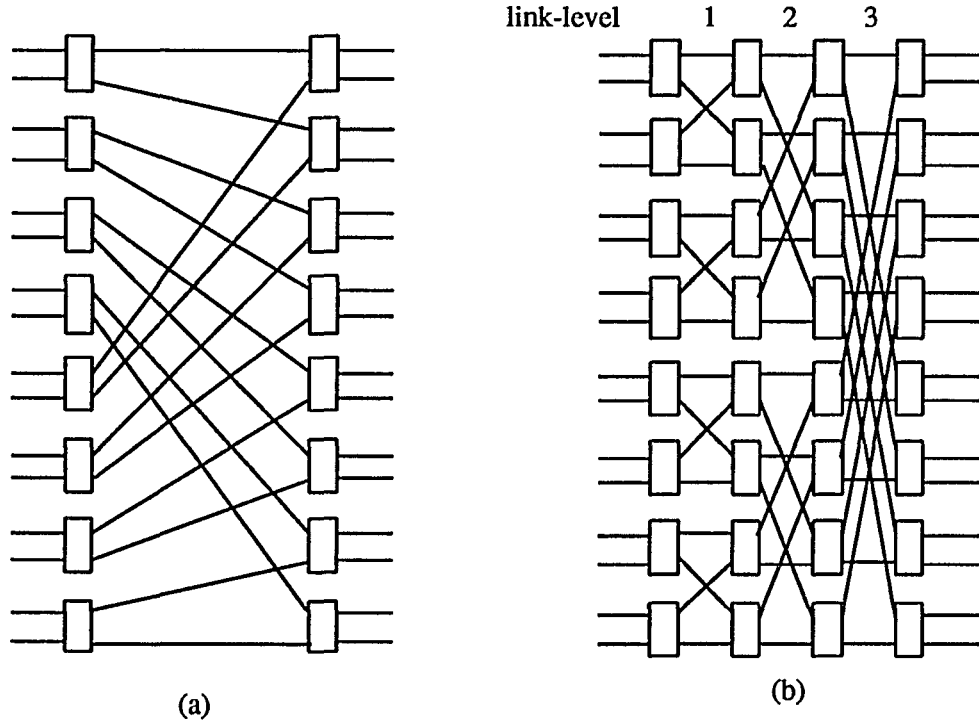


Figure 3.5. (a) Shuffle permutation σ and (b) Banyan permutation r .

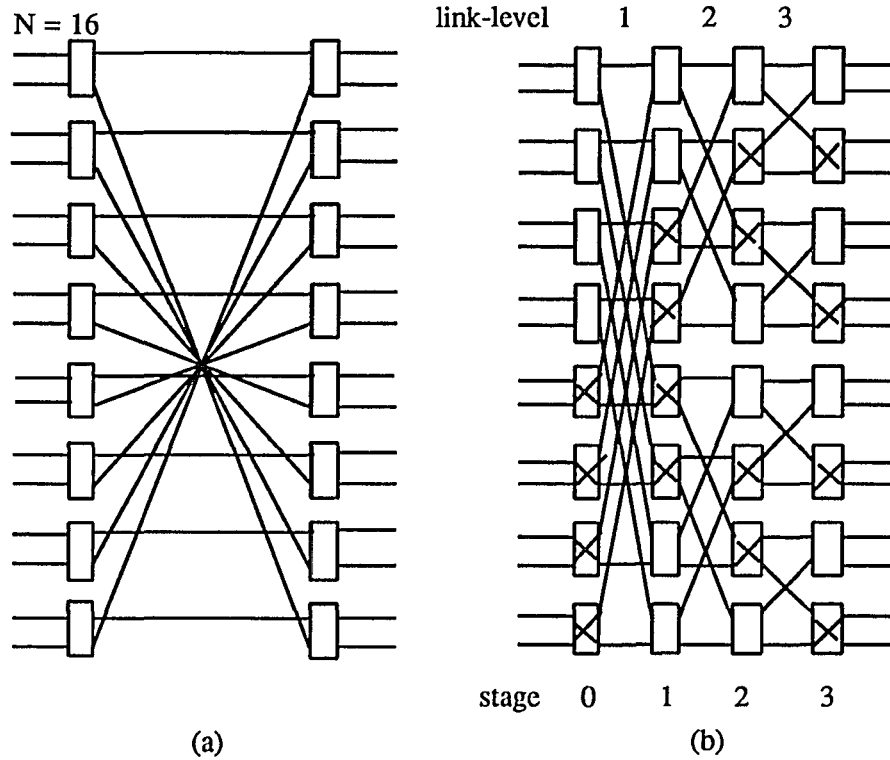


Figure 3.6. (a) Balanced permutation τ and (b) permutation ψ of the modified data manipulator.

Therefore, we know that a shuffle permutation of N inputs can be replaced by a Banyan network with $\log_2 N$ stages and every switching element in a stage is in the straight state. Now we can see that the sorting networks with complex connections can be replaced by equivalent networks which are inherently easier to reconfigure. Each switching element in the equivalent networks is either in the bypass state or in the exchange state and therefore, is very simple to implement such that the area and time penalty is negligible compared with the entire sorting network. The number of cells used in the modified equivalent networks is equal to that of the original networks and the time latency is not changed. Therefore, the two multi-way mergers can be replaced by a more regular and reconfigurable equivalent multi-way merger which introduces little time overhead.

3.5. Defect Tolerance

A. Bottom-Level and Top-Level Reconfigurable Structures

Since the odd-even sorter is used in both the bottom-level and the top-level of the sorting network, the reconfigurable structure will be the same and therefore, we use an example 6-stage odd-even sorter in Fig. 3.7 to illustrate our approach. In this section, a cell can be a bottom cell or a top-level module. Two redundant cells are added to each stage with one at the bottom and the other at the top of the stage. Two switching elements are associated with each cell to control input/output to and from the cell. The input and output functions of a switching element

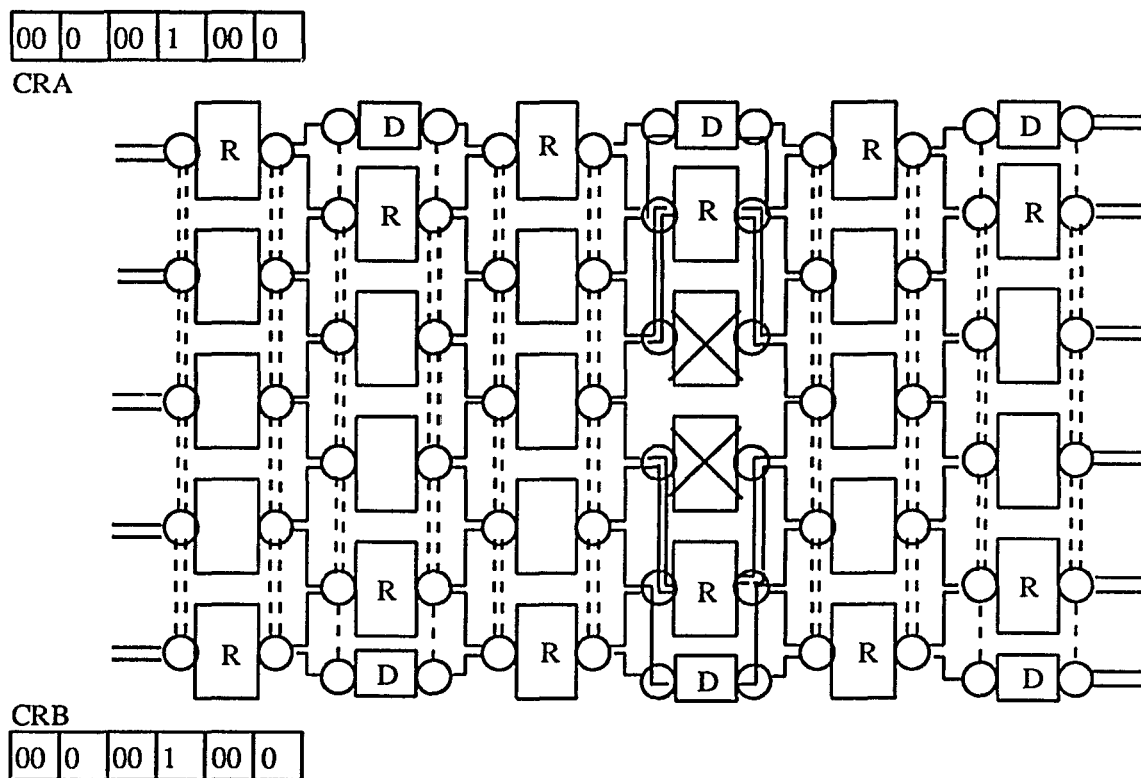


Figure 3.7. Example reconfigurable structure of the odd-even transposition sorter.

are shown in Fig. 3.8 which are controlled by the reconfiguration control registers (*CRA* and *CRB*) where *S* is a control bit for a cell. Each field in a reconfiguration control register controls a stage. The number of bits in each field for a stage with *l* cells is *r* (where $2^r \geq l-2$). Cells in a stage are numbered from 1 to *l*. The bit patterns in *CRA* and *CRB* determine whether a faulty cell will be replaced by a redundant cell in the top row or the bottom row. Table 3.6 illustrates the meaning of a field in *CRA* and *CRB*. The switches of a delay cell is activated when its neighbor cell in the same stage is activated. For example, in Fig. 3.7 there are two faulty cells in stage 4 and the corresponding switch settings after reconfiguration are *CRA*=1 and *CRB*=1, i.e., switches of cells 1 and 2 in this stage are activated by *CRA*, switches of cells 3 and 4 are activated by *CRB*. Two switches are also associated with each "D" registers, one is a 2-to-1 multiplexer and the other is a 1-to-2 demultiplexer. This structure can tolerate up to two faulty cells in each stage.

It has been shown shown in [74] that by adding two extra stages in an odd-even transposition sorter as shown in Fig. 3.9, any single fault which makes a sorting cell perform an incorrect swap (we call this a functional fault) can be recovered automatically. In addition to

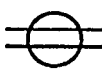







	CRA		CRB	
switch	S=0	S=1	S=0	S=1
input port				
output port				

Figure 3.8. Output functions of a switching element in the top-level.

Table 3.6. Regions covered by *CRA* and *CRB*.

$l=5(r=2)$			$l=4(r=1)$		
	<i>CRA</i>	<i>CRB</i>		<i>CRA</i>	<i>CRB</i>
00	normal	normal	0	normal	normal
01	cells 1-2	cells 3-5	1	cells 1-2	cells 3-4
10	cell 1-3	cell 4-5			

the fault masking property for the single functional fault, each sorting cell in Fig. 20 includes bypass registers such that the faulty cells generating nonfunctional errors can be bypassed without affecting system synchronization. The bypassed cells can be viewed as faulty cells which do not swap at all, and these errors will be recovered by the two extra stages [74]. Therefore, the odd-even transposition sorter can tolerate up to two faulty stages by simply bypassing the faulty cells without the need to restructuring the entire sorter.

Since the sorting network is a pipelined structure, increasing the number of extra stages will increase the pipeline latency only. After the pipeline has been filled, a set of N outputs will

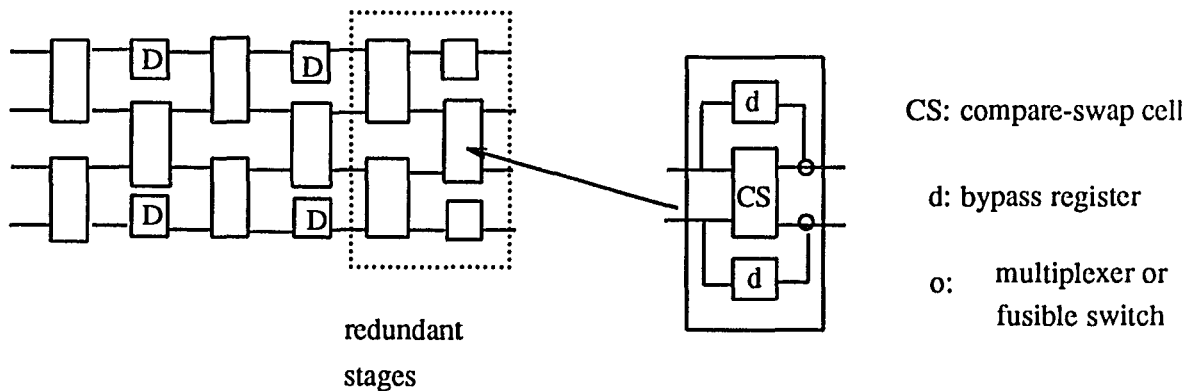


Figure 3.9. On-line reconfigurable odd-even transposition sorter.

be generated at every clock period. For the top-level sorter, since the bottom-level sorter in each submodule can be bypassed, each module can also be bypassed by setting all sorters in the submodules of this module bypassed. Therefore, the same reconfigurable structure with redundant rows and columns of modules or cells can be applied to both levels.

B. Middle-Level Reconfigurable Structure

However, with clustered defects, it is possible that there are more than two faulty stages in a bottom-level sorter. If the number of faults in a bottom-level sorter is larger than the amount of redundancy it has or if any physical defect causes a faulty cell unable to be bypassed, this sorter will be declared as unrepairable and switched out by the reconfiguration scheme described in the following and replaced by a redundant submodule.

Input lines and output lines of a submodule in this level are connected to three submodules in the preceding stage and succeeding stage, but not all of them are the nearest neighbors of that submodule. Each submodule has two switches, one in the input port to select two out of three inputs and the other in the output port to direct data to two of the three output lines. It should be noted that after the transformation to the equivalent network, the shuffle interconnections in the bitonic merger are replaced by the Banyan permutation and therefore, the bitonic merger is now like a modified data manipulator. The *Omega* network in a sorting block of the balanced merger and the τ permutation between sorting blocks are replaced by the modified data manipulator. Therefore, the balanced merger is now connected by a series of modified data manipulators.

The reconfigurable structure of the multi-way balanced merger is shown in Fig. 3.10(a). In Fig. 3.10(b) we show an example after reconfiguration. The reconfiguration strategy in Fig. 21 includes a redundant row of submodules which can either the bottom row or the top row.

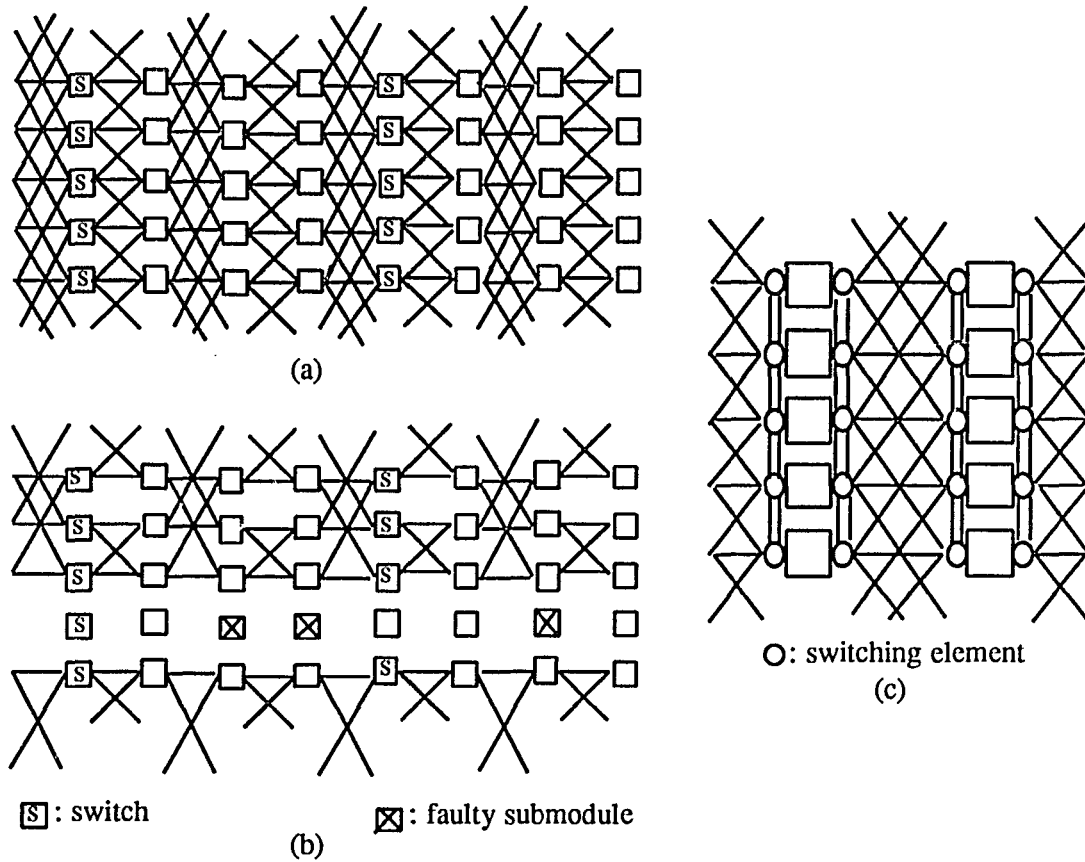


Figure 3.10. Example reconfiguration in the middle-level.

Note that the switching elements in the permutation networks should also be reprogrammed since the setting of switching elements in the equivalent network to perform the τ permutations are different and depend on the positions of these switching elements. Similarly, after restructuring the system, a sorting element in the equivalent multi-way bitonic merger should be reprogrammed according to its position in the new structure due to the nonuniform cell structure in the bitonic merger.

However, the reconfigurable structure in Fig. 3.10(a) can tolerate faulty submodules in the same row only, any two faulty submodules in two different rows will cause the submodule

unrepairable. This problem can be solved by modifying the two switches at the input and output ports as shown in Fig. 3.10(c). Excluding a faulty submodule and replacing by a redundant submodule in the same column can be completed with the same control scheme as that in Fig. 3.7 and the same switch functions as those in Fig. 3.8.

The major drawback of the defect tolerant multi-way merger in Fig. 3.10 is that the reconfigurable butterfly interconnections between two stages have wrap-around connections. An example butterfly interconnection between two stages is shown in Fig. 3.11(a), where each stage has $k=8$ sorting elements and one redundant element ($R=1$). If each element has $n=3$ outputs with $m=1$ bit line per output, then there will be 8 $(=(k+r-1)mn/3)$ wrap-around interconnections. Let a represent the wire width of an interconnection and b the space between two interconnections as shown in Fig. 3.11. Then the distance between two stages in a butterfly interconnections is $k/(2\times\sqrt{3})\times(h+b)$ since $d_1=d_2$. The length on the longest interconnection due to wrap-around will be about $(k/2+k+1)\times(h+b)$ which is $3\times\sqrt{3}$ times longer than the shortest interconnection. Therefore, in addition to increasing the wiring complexity, the wrap-around interconnections will slow down the system clock significantly and thus, reduce the system throughput.

However, this drawback can be avoided by replacing the wrap-around interconnections with interconnections directly from the source to the destination. Let the submodules in a stage be numbered from 0 to k . As shown in Fig. 3.11(b), the three wires of a sorting element s ($0\leq s\leq k$) connect respectively to the sorting elements s , $k/2$, and $k/2+1$ in the next stage, if $s\leq k/2$, otherwise, they connect to the sorting elements s , $s-k/2$ and $s-k/2-1$ in the next stage. The configuration of Fig. 3.11(b) is then equivalent to Fig. 3.11(a). The length of the longest interconnection in Fig. 3.11(b) becomes $2(k/2+1)(h+b)/\sqrt{3}$ which is approximately $2/(3\sqrt{3})$ that

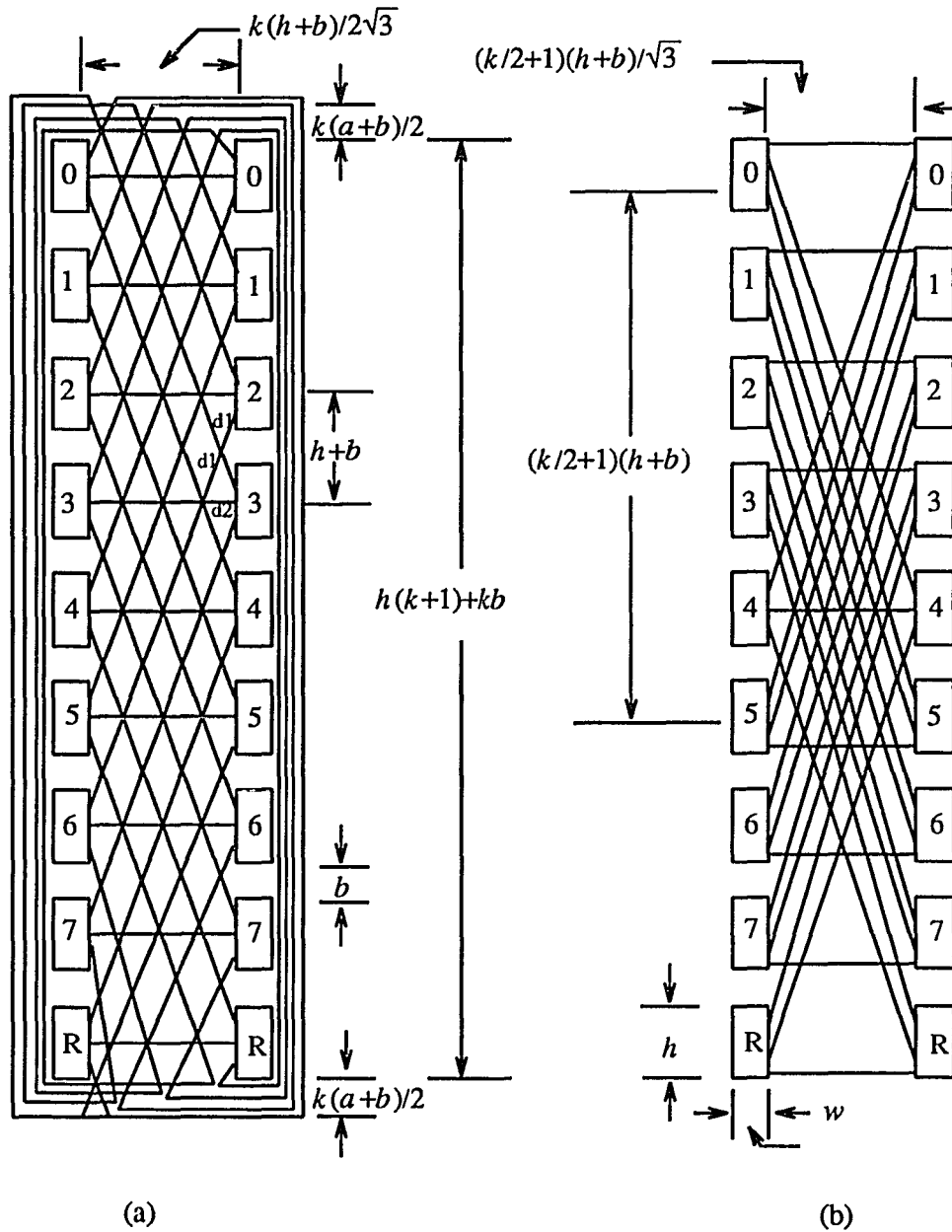


Figure 3.11. Butterfly interconnections with and without wrap-around wires.

of the longest interconnection in the wrap-around structure.

The advantage of Fig. 3.11(b) over Fig. 3.11(a) in the total interconnection area is illustrated as follows. The total wire area for the original butterfly interconnections can be derived

to be

$$A_{b-o} = [(k+1) \cdot h + k \cdot b + k \cdot (a+b)] \times \left[\frac{k}{2} \cdot \frac{(h+b)}{\sqrt{3}} + k \cdot (a+b) \right]$$

and the total wire area of the modified butterfly is

$$A_{b-m} = [(k+1) \cdot h + k \cdot b] \times \left[\left(\frac{k}{2} + 1 \right) \cdot \frac{(h+b)}{\sqrt{3}} \right]. \quad (3.7)$$

Since $A_{b-o} > A_{b-m}$, not only the wrap-around connections are eliminated, but also the area is reduced.

3.6. Yield Analysis

The yield of a WSI array processor is defined as the probability that during the manufacturing process, defects are distributed into cells, switches, and wires of the array in such a way that all defective elements can be tolerated [75]. In order to evaluate the improvements on yield after redundancies are introduced in each level, yield modeling and analysis are developed in this section. Our approach is to start the analysis at a stage of the bottom-level odd-even sorter.

If the defects are randomly distributed, the yield $Y = e^{-DA}$ where D is the average defect density and A is the area of the chip or wafer. However, in the real manufacturing environment, the defects have a tendency towards clustering. Therefore, the yield Y follows the more accurate negative binomial distribution, *i.e.*, $Y = (1 + DA/\alpha)^{-\alpha}$ [76]. α is a parameter representing the level of clustering, which usually takes a value around 1 or 2 [77]. The probability of having k defects in a stage is then [76]

$$P_w(k) = \frac{\Gamma(k+\alpha) \left(\frac{DA}{\alpha} \right)^k}{k! \Gamma(\alpha) \left(1 + \frac{DA}{\alpha} \right)^{\alpha+k}}.$$

The yield Y_{bs} of a stage at the bottom-level with $n-r$ normal cells and r redundant cells can

then be derived as

$$Y_{bs} = \sum_{k=0}^r \frac{\Gamma(k+\alpha) \left(\frac{DA}{\alpha}\right)^k}{k! \Gamma(\alpha) \left(1 + \frac{DA}{\alpha}\right)^{\alpha+k}} \times P_{knr}. \quad (3.8)$$

where P_{knr} is the reconfiguration coverage which represents the probability of successfully reconfiguring a stage of n cells with r redundant cells and k defects. For our reconfiguration scheme, $P_{knr}=1$ if $k \leq r$, and in order to simplify the analysis we will assume that $P_{knr}=0$ for $k > r$ to obtain the lower bound of Y_{bs} . Actually, our scheme can tolerate more than k defects if some of the defects fall on the same cell.

Defective wires or switches between two stages can also be tolerated by using alternative paths and bypassing the corresponding cell which can be viewed as a faulty cell and replaced by a redundant cell. Therefore, the effect of defects in wires and switching elements on the yield can be included into the model Y_{bs} by adding the area of switches and interconnections to the total area A in equation (3.8). This is different from the model in [78] where the effect of wires on the yield is calculated independently because no faulty interconnection can be tolerated.

Therefore, the yield of a bottom-level sorter, Y_b , is equal to Y_{bs}^m , if there are m stages and no redundant stage in the sorter. If two redundant stages are included, the bottom-level sorter can tolerate up to two consecutive faulty stages, and the yield is

$$Y_b = Y_{bs}^{m+2} + (m+2)Y_{bs}^{m+1}(1-Y_{bs}) + (m+1)Y_{bs}^m(1-Y_{bs})^2.$$

Since a submodule is an odd-even sorter, the yield of a stage in the middle-level sorter is

$$Y_{is} = Y_b^{i+2} + (i+2)Y_b^{i+1}(1-Y_b) + (i+2)(i+1)Y_b^m(1-Y_b)^2/2$$

assuming there are i submodules plus two redundant submodules in each stage.

Let A_{iw} represent the area of the interconnections and the switching elements between two neighbor stages of a middle-level sorter as shown in Fig. 3.10(c). The yield Y_{iw} of the

interconnection area can be derived by using equation (8). The yield of a stage including the interconnection area is then $Y_{is}' = Y_{is} \times Y_{iw}$. Therefore, the yield of a middle-level sorter, Y_i , will be $(Y_{is}')^q$ if there are q stages.

Similarly, the yield, Y_{ts} , of a stage in the top-level sorter is

$$Y_{ts} = Y_t^{j+2} + (t+2)Y_t^{j+1}(1-Y_t) + (j+2)(j+1)Y_t^m(1-Y_t)^2/2$$

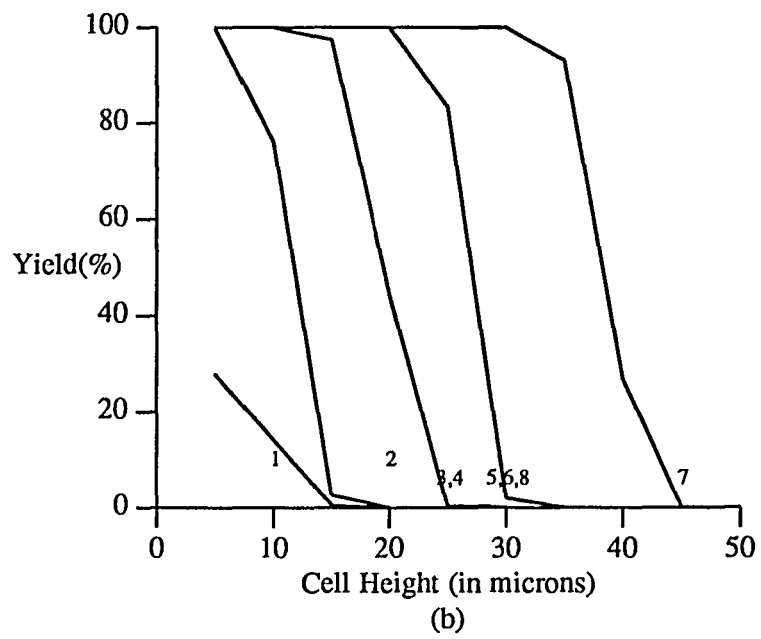
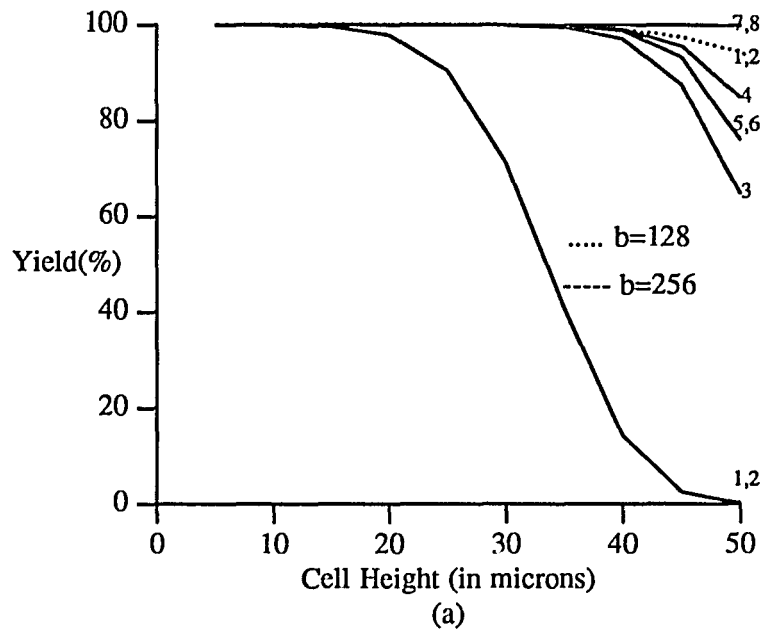
if there are j rows plus two redundant rows in each stage and $Y_t = Y_{ts} \times Y_{tw}$ where Y_{tw} is the yield of the interconnection area between two top-level stages. Then, Y , the yield of the top-level sorter (*i.e.*, the yield of the *HMSN*) is

$$Y = Y_t^{l+2} + (l+2)Y_t^{l+1}(1-Y_t) + (l+1)Y_t^l(1-Y_t)^2, \quad (3.9)$$

if there are l stages plus two redundant stages in the top-level odd-even sorter.

An example sorting cell in [74] is used in the following simulation to evaluate how much yield improvement can be achieved by various redundancies in each level. The height of a cell is assumed to be between 5 μm (micrometers) and 50 μm . From equation (3.7), the area A_{iw} of the butterfly interconnections in the middle-level is proportional to $(kh)^2$ where $k = b_{\max}$ and therefore, the larger the A_{iw} is the smaller the Y_{iw} will be. Since $Y_{is}' = Y_{is} \times Y_{iw}$, any small decrease of the value of Y_{iw} will reduce the value of Y_{is}' and thus, drop the yield of the middle-level sorter significantly. This is due to the fact that $Y_i = (Y_{is}')^q$, where q is the number of stages in a sorter at this level and is no less than 64 (*i.e.*, $b_{\max} = 128$).

Yield with respect to p for the example cases in Table 3.1 are shown in Fig. 3.12 (a), (b) and (c). The defect density D in the cell area is assumed to be two defects per cm^2 and α is 2. However, since the wires and switches are much simpler and more regular than the cells, they are less vulnerable to defects and hence we assume that defect density in the interconnection area is one tenth of D . If no redundancy is included in an *HMSN*, the yield is zero for all cases



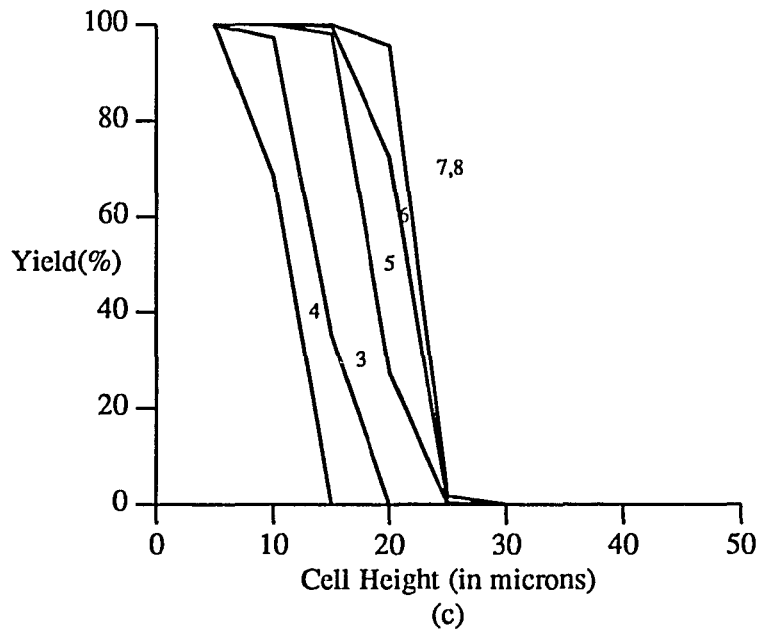


Figure 3.12. Example yield analysis.

in Tables 3.2-3.5. From Fig. 23(a), we see that the *HMSN* performs well in every case with $p=20$ and $b_{\max}=256$ (the solid lines) for cell heights less than 25 μm . The dotted line in Fig. 3.12 (a) shows the yield when $b_{\max}=128$. We only show cases 1 and 2 which have about 92% yield even if the cell height = 50 μm . For cases 3 to 8, the yields are always close to 100%.

In Fig. 3.12 (b), $b_{\max}=256$ and $p=100$, the yield drops quickly when only one redundant row is incorporated in every level for case 1. Also, we can see that case 7 performs better than case 8 even case 8 has two more redundant columns in the bottom-level sorter. The reason is that when two redundant rows are included in the bottom level, the yield of a bottom level sorter will be almost 1, and two more redundant columns cannot generate any further significant improvement on yield. Therefore, the difference on yield between case 7 and case 8 depends on the value p_1 . Since A_{iw} grows with $(p_1 \times \text{cell-height})^2$, a difference in p_1 can generate a large difference in Y_{iw} . From Table 3.3, since $p_1=10$ for case 7 and $p_1=20$ for cases 6 and 8, case 7

has a higher yield than cases 6 and 8. The yield of case 5 with $p_1=25$ is less than that of case 6, but it is not significant. The same reason can be applied for the differences among cases 2, 3, and 4.

If $p=200$ and $b_{\max}=256$, the yield does not improve for cases 1 and 2, and for other cases the yield drops sharply when the cell height increases past a certain value as shown in Fig. 3.12(c). For example, let us look at case 7 with a cell height of 20 or 25 μm , respectively. Our simulation results show that Y_{is} is 1 for both heights and $Y_{iw}=0.999$ if height=20 μm and $Y_{iw}=0.997$ if height=25 μm . However, since $Y_i=(Y_{iw})^{81}$ for this case, we have $Y_i=0.9527$ if height=20 μm and $Y_i=0.8495$ if height=25 μm . The two redundant rows in the top level make $Y_{is}=0.983$ if height=20 μm and $Y_{is}=0.73$ if height=25 μm and therefore, we have $Y=0.9559$ and $Y=0.017$, respectively. This case shows how the wire complexity can decrease the yield so significantly when both b_{\max} and p_1 are large.

3.7. Summary

A novel hierarchical modular sorting network is presented in this chapter. It uses less hardware and converges faster than a single-level odd-even transposition sorter and the wire complexity problem of the bitonic sorter in VLSI or WSI is alleviated. A cost function is derived to determine the optimal sorting capability at each level and minimize the hardware complexity when redundancy is provided at every level of the hierarchy. The hierarchical sorting network is very regular in structure and hence easier to reconfigure than any existing sorting network with the same time complexity. Hierarchical reconfiguration strategy is proposed to tolerate the defective elements in an efficient manner. Detailed yield analysis is performed on the hierarchical sorting networks. Yield improvements for cases with various number of spares

are evaluated. The simulation results show that the defect tolerant *HMSN* achieves a significant yield increase over a nonredundant sorting network.

CHAPTER 4.

THE TRAPEZOID SORT

4.1. Introduction

Parallel sorting algorithms for two-dimensional mesh-connected processor arrays have been intensively studied in [16, 15, 14] and more recently, in [19, 21, 3]. Efficient implementations of these sorting algorithms in two-dimensional VLSI models are shown in [33, 60]. One of the earliest results on sorting rectangular arrays of numbers was presented by Thompson and Kung [14]. By mapping the odd-even merge sort and the bitonic sort [6] onto an $n \times n$ mesh-connected array, it takes $(6n + O(n^{2/3} \log_2 n))t_r + (n + O(n^{2/3} \log_2 n))t_c$ and $(14(n-1) - 8 \log_2 n)t_r + (2 \log_2^2 n + \log_2 n)t_c$, respectively, to sort n^2 data items. The notation t_r represents the time for routing a value in a processor to one of its neighbor processors and t_c represents the time for comparing two values. Their efforts were improved by Nassimi and Sahni [15], and Kumar and Hirschberg [16] with improved constant factors. These earlier efforts were adaptations of inherently parallel algorithms such as the odd-even merge sort and the bitonic sort to the mesh-connected array in an efficient manner such that the time complexity is $O(n)$. However, these implementations spend most of the time in routing data to appropriate processors, and the complicated data movements in successive iterations result in complicated control structures and thus, offset the advantage of simple interconnections.

Recently, Sado and Igarashi [19], and Scherson and Sen [21] presented two similar paral-

lel sorting algorithms independently, the parallel bubble sort and the shear sort, respectively, for two-dimensional SIMD model. These sorting algorithms are based upon a repeated application of the bubble-sort method [5] to the rows and columns of the array to be sorted. In this chapter we will refer either of these two as the *row-column sort* algorithm since it consists of two basic operations: the row sort and the column sort. It is a true two-dimensional sorting technique and has the advantages that it is extremely simple to implement in any of the two-dimensional computing models and its control complexity is reduced considerably due to its repetitive and nonrecursive nature. Its major drawback is that it requires $\lceil \log_2 n \rceil + 1$ iterations to sort an $n \times n$ input sequence where each iteration includes one row sort and one column sort.

In section 4.2, we present a new two-dimensional sorting algorithm, the *trapezoid sort*, which preserves the properties of simple control hardware and ease of implementation of the *row-column sort*, and the complexity is improved to $\lceil \log_2 l \rceil + 1$ iterations with $l \approx \sqrt{n}$. Further analysis in section 4.3 gives the proof of convergence and some special properties of the algorithm.

4.2. The Trapezoid Sort

Let $Q=[Q_{ij}]$ be an $n \times n$ mesh-connected array of identical processors as in [79], onto which we have mapped an input sequence S . Sorting the sequence S is then equivalent to sorting the elements of Q in some predetermined indexing scheme. Here, we use the *snake-like row major (SLRM)* indexing scheme as shown in Fig. 4.1 to order a two-dimensional array of elements.

1	2	3	4	5	6	7
14	13	12	11	10	9	8
15	16	17	18	19	20	21
28	27	26	25	24	23	22
29	30	31	32	33	34	35
42	41	40	39	38	37	36
43	44	45	46	47	48	49

Figure 4.1. An example of snake-like row major indexing scheme.

The *trapezoid sort* is presented in Fig. 4.2 with *PASCAL*-like notations. In this procedure, the i th row and the j th column of the matrix Q are denoted by $Q[i, 1 \dots n]$ and $Q[1 \dots n, j]$, respectively. A row vector is sorted in *nondecreasing order* from left to right by the procedure *row-sort* and a column vector is sorted in *nondecreasing order* from top to bottom by the *column-sort* procedure. Both the *row-sort* and the *column-sort* procedures are implemented based on the odd-even transposition sort [5] which compares two neighbor values in a regular and alternating manner. The procedure $\overline{\text{row-sort}}$ sorts a row vector of Q in an opposite way to the *row-sort* procedure, i.e., *nonincreasingly* from left to right. The parameter l which will be determined in the next section is used to control the number of iterations required of the *row-column sort* module (step 4) in the procedure to obtain a *snake-like row major* ordered output sequence.

The *trapezoid sort* can be decomposed into five steps. In the first step, all rows are sorted in the same direction from left to right and then these sorted rows are shifted right cyclicly by $(t-1)$ elements from row $t=1$ to n in step 2. An example output right after steps 1 and 2 are executed is shown in Fig. 4.3, where the values 1 to 7 are randomly distributed in each row of the 7×7 matrix initially. After step 2 has been executed, the values 1, 2, 3, ..., 7 are also distributed uniformly in each column. The configuration in Fig. 4.3 will be referred as Q' in

```

Procedure Trapezoid Sort ( $Q, l$ );
begin
  for all  $t := 1$  to  $n$  do in parallel    /* step 1 */
    row-sort  $Q[t, 1 \dots n]$ ;
  for all  $t := 1$  to  $n$  do in parallel    /* step 2 */
    cyclic shift right  $Q[t, 1 \dots n]$  by  $(t-1)$  positions;
  for all  $t := 1$  to  $n$  do in parallel    /* step 3 */
    column-sort  $Q[1 \dots n, t]$ ;
  for  $i := 1$  to  $\lceil \log_2 l \rceil$  do      /* step 4 */
    begin
      for all  $t := 1$  to  $n$  do in parallel
        if odd( $t$ )
          then row-sort  $Q[t, 1 \dots n]$ 
          else row-sort  $Q[t, 1 \dots n]$ ;
      for all  $t := 1$  to  $n$  do in parallel
        column-sort  $Q[1 \dots n, t]$ ;
    end;
  for all  $t := 1$  to  $n$  do in parallel    /* step 5 */
    if odd( $t$ )
      then row-sort  $Q[t, 1 \dots n]$ 
      else row-sort  $Q[t, 1 \dots n]$ ;
end.

```

Figure 4.2. The trapezoid sort algorithm.

1	2	3	4	5	6	7
7	1	2	3	4	5	6
6	7	1	2	3	4	5
5	6	7	1	2	3	4
4	5	6	7	1	2	3
3	4	5	6	7	1	2
2	3	4	5	6	7	1

Figure 4.3. An example output of step 2 with $n=7$.

section 4.3 to describe the orders of rows after the cyclic shift right operation. Step 3 is a column-sort to sort values of each column nondecreasingly. The statements in steps 4 and 5 together can be viewed as an independent module (row-column sort module) which is equivalent to the *shear sort* in [21] or the *parallel bubble sort* in [19] except that it requires less iterations of the row sort and the column sort.

4.3. Analysis and Time Complexity

Let i represent the number of iterations of the row-column sort which has been executed on Q , Q^i represent the matrix after i iterations of the row-column sort (a row sort followed by a column sort) except that a cyclic shift operation is performed between the row sort and the column sort in the first iteration (steps 1 to 3). Properties of the *trapezoid sort* and the amount of improvement on the number of iterations over the suboptimal *shear sort* and *parallel bubble sort* will be analyzed by applying the *zero-one* ($\{0-1\}$) *principle* [5]. For completeness, the $\{0-1\}$ principle is restated in Theorem 4.1.

THEOREM 4.1: If a network with n input lines sorts all 2^n sequences of 0's and 1's into nondecreasing order, it will sort any sequence of n numbers into nondecreasing order. \square

For the purpose of applying the {0-1} principle, Q is assumed to contain only 0's and 1's and a row is said to be *clean* if it contains identical elements, *i.e.*, only 0's or only 1's, otherwise it is *dirty*. In Theorem 4.2, we will show that the maximum number of dirty rows after the first iteration of row-column sort can be determined by an equation similar to that of calculating the area of a trapezoid. This is the reason why we call it the *trapezoid sort*. By using the {0-1} principle, we will prove in Theorem 4.3 that the *trapezoid sort* is a complete sorting algorithm with the variable " l " in step 4 being equal to the maximum number of *dirty rows* after the first iteration when there are initially n zeros in the matrix Q .

A. Finding the Maximum Number of Dirty Rows

In our analysis, the relationship between row j and row $j+1$ in Q^i will be obtained first in Lemma 4.1 based on the number of zeros in Q^i . A method is then derived in Theorem 4.2 to find the maximum number of dirty rows in the matrix Q after the first iteration of the row-column sort.

LEMMA 4.1: Let the number of zeros in row j after i iterations be represented by $\gamma_j(i)$. For all $j, k, 1 \leq j < k \leq n$, the number of zeros in the j th row of Q^i is no less than that in the k th row.

PROOF: If the number of zeros in row j of Q^i is less than that of row k , then there will be at least a zero in $Q^i[k, m]$ and a one in $Q^i[j, m]$, $1 \leq m \leq n$. This is in contradiction to our assumption that all columns are sorted in nondecreasing order after a column sort. Thus, the zero in $Q^i[k, m]$ should pop up to $Q^i[j, m]$. Therefore, for all $j, k, 1 \leq j < k \leq n$, there are at least as many zeros in the j th row as in the k th row. That is, $\gamma_j(i) \geq \gamma_k(i)$, for all $1 \leq j < k \leq n$. \square

In step 5 of the *trapezoid sort* algorithm, a final row sort is used to sort the output sequence into the *SLRM* order after all elements are in their final row positions following step 4

(this will be shown in Theorem 4.3). That is, if the sorting algorithm can sort an input sequence with pn zeros or $(p+1)n$ zeros, then it can sort any input sequence with the number of zeros between pn and $(p+1)n$. Therefore, without loss of generality, in the following analysis we will consider input sequences with pn zeros only where p can be any integer and $p \leq n$.

THEOREM 4.2: Let the maximum number of dirty rows in Q^i be $l_p(i)$. If there are initially pn zeros in Q , the relationship between the maximum number of dirty rows in Q^1 and pn will follow an equation similar to that of calculating the area of a trapezoid. That is, $\frac{l_p(1) \cdot [l_p(1)+1]}{2} + r_p = pn$, or equivalently, $\frac{l_p(1) \cdot [l_p(1)+1]}{2} \leq pn < \frac{[l_p(1)+1] \cdot [l_p(1)+2]}{2}$. The number of remaining zeros which do not increase the height of the trapezoid is represented by r_p ($\leq l_p(1)$) in the above equation.

PROOF: Since the operation of the first column sort (*i.e.*, step 3 of Fig. 4.2) is just to move zeros in each column of Q' to the top of Q^1 , the maximum number of dirty rows in Q^1 can be regarded as the maximum number of zeros allowed in any column of Q' .

To have a zero located at $Q'[l, m]$, from Fig. 4.3 we know that at least $n - [l - (m+1)]$ and $m - l + 1$ zeros are required in row l for $l > m$ and $l \leq m$, respectively. For Q' to have the maximum number h of dirty rows with the least total number of zeros, there should be h zeros in the same column and in continuous rows, *i.e.*, located in $Q'[l, l]$, $Q'[l-1, l]$, $Q'[l-2, l]$, ..., $Q'[l-h+1, l]$ for $l-h \geq 0$ or in $Q'[l, l]$, $Q'[l-1, l]$, ..., $Q'[1, l]$, $Q'[n, l]$, $Q'[n-1, l]$, ..., $Q'[n-(h-l), l]$ for $l-h < 0$. Therefore, with a total of $1+2+\dots+h$ zeros and arranged in the matrix Q' according to the above restrictions (*i.e.*, topologically equivalent to a trapezoid), we can have the maximum number of h dirty rows in Q' by having the least number of total $\frac{h(h+1)}{2}$ zeros.

In the case of having pn zeros in Q , h will be equal to $l_p(1)$ and $\frac{l_p(1)[l_p(1)+1]}{2} + r_p = pn < (p+1) \cdot n$. That is, $l_p^2(1) + l_p(1) - 2(p \cdot n - r_p) = 0$ and $l_p^2(1) + l_p(1) - 2[(p+1) \cdot n - r_p] < 0$. In other words, $l_p(1) = \frac{-1 + \sqrt{1 + 8(p \cdot n - r_p)}}{2}$. \square

An example with $p=1$ is shown in Fig. 4.4, where $n_1=7$, $l_1(1)=3$, $\gamma_1(1)=4$ and $r_1=1$. Also, from Fig. 4.4 we can see that with the maximum number of dirty rows in Q^1 or Q' , the number of zeros in row 1 of Q^1 will be at least as large as the maximum number of dirty rows (i.e., $\gamma_1(1) \geq l_p(1)$).

B. Proof of Convergence

From Corollary 1 in [8], we know that the number of iterations in the row-column sort for any 0/1 input sequence is determined by the number of dirty rows initially in the matrix. For example, if there are initially d dirty rows in the matrix, then $\lceil \log_2 d \rceil + 1$ iterations are required to sort the sequence. Based on this property and Theorem 4.2, counting the maximum number of dirty rows in the array after step 3 has been executed is equivalent to that of calculating the maximum height of a trapezoid which can be generated by the input sequence. Therefore,

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	0	0	0	0	1
1	1	1	0	0	1	1
1	1	1	1	0	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

Figure 4.4. The maximum number of dirty rows = 3 for $Q_{7 \times 7}$.

the maximum number of iterations of the row-column sort to sort any input sequence is equal to $\lceil \log_2(n-1) \rceil + 1$ since the maximum number of dirty rows after the first row-column sort (steps 1 to 3) is equal to $n-1$ which is generated by input sequences with $n(n-1)/2$ or more zeros. However, in the following theorem we will prove that any input sequence can be sorted by the trapezoid sort algorithm with the maximum number of iterations equal to $\lceil \log_2 l_1(1) \rceil + 1$, where $l_1(1) \approx \sqrt{n}$ is the maximum number of dirty rows which can be generated by step 3 for an input sequence with n zeros.

THEOREM 4.3: The *trapezoid sort* is a complete sorting algorithm.

PROOF: In the following analysis, we will first prove that the trapezoid sort algorithm can sort any input sequence with n zeros, and then prove that if this sorting algorithm can sort an input sequence with n zeros, it can sort any input sequence with up to n^2 zeros.

The process of convergence in sorting the matrix Q with n zeros can be regarded as a process of recursive merging. Let $h=l_1(1)$. In order to have the maximum number of dirty rows in the matrix, we have $\gamma_1(1) \geq \gamma_2(1) \geq \dots \geq \gamma_h(1)$, and $\gamma_j(1)=0$ for all $j>h$. Since there are n zeros in the matrix, $\gamma_1(1)+\gamma_2(1)+\dots+\gamma_h(1)=n$ and $\gamma_j(1) \neq 0$ for $1 \leq j \leq h$. In addition, from the proof of Theorem 4.2 we have $\gamma_1(1)+\gamma_2(1) < n$, $\gamma_3(1)+\gamma_4(1) < n$, ..., $\gamma_{n-1}(1)+\gamma_n(1) < n$. Therefore, after the second row sort, all zeros in $\gamma_{2b-1}(1)$ are packed at the left and all zeros in $\gamma_{2b}(1)$ are packed at the right ($1 \leq b < n/2$). Then, all zeros in $\gamma_{2b}(1)$ are moved up b rows and so are all zeros in $\gamma_{2b-1}(1)$ after the second column sort and therefore, two dirty rows, j and $j+1$, are merged together.

According to Lemma 4.1 and Theorem 4.2, for an input sequence with n zeros and maximum number of dirty rows, $\gamma_{2b-1}(i)+\gamma_{2b}(i)$ should be less than n , unless after the i th row sort all zeros are moved to row 1. After each repetition of the row-sort in step 4, since all zeros in

an odd row are moved to the left and all zeros in an even row are moved to the right, $\gamma_{2b-1}(i)$ and $\gamma_{2b}(i)$ will be combined together to form a new row. That is, after each row-column sort, half of the remaining dirty rows become clean. Then, after i iterations of the row-column sort, $l_1(i)$ will not be greater than $\left\lceil \frac{l_1(i-1)}{2} \right\rceil$. Therefore, in addition to the first iteration of the row-column sort, $\left\lceil \log_2 l_1(1) \right\rceil$ iterations are required to clean the dirty rows and one last row sort is required to order zeros in the same row. Hence, the *trapezoid sort* can sort any input sequence with n zeros.

Assume that the input sequence with pn zeros has the maximum number of $n/2$ dirty rows, i.e., $l_p(1)=n/2$. Then, $\frac{n}{2}+k \leq l_{p+k}(1) \leq \frac{n}{2}+2k$ if there are kn more zeros in the sequence. From Lemma 4.1 and Theorem 4.2, we have $\gamma_1(1) \geq \gamma_2(1) \geq \dots \geq \gamma_{2k+1}(1)$, and $\gamma_1(1) \geq \frac{n}{2}+k$, $\gamma_2(1) \geq \frac{n}{2}+k-1$, ..., $\gamma_{k-1}(1) \geq \frac{n}{2}$, ..., $\gamma_{2k+1}(1) \geq \frac{n}{2}-k$. After the second row-column sort, $\gamma_1(1)$ and $\gamma_2(1)$ will be merged as $\gamma_1(2)$, $\gamma_3(1)$ and $\gamma_4(1)$ will be merged as $\gamma_2(2)$, and so on. Therefore, we have $\gamma_1(2) \geq n+2k-1$, $\gamma_2(2) \geq n+2k-5$, ..., $\gamma_{t+1}(2) \geq n+2k-4t-1$. For $k=1$, since $n+2-1 > n$, we will have one clean row after the second row-column sort. For $k=2$, although $n+2k-5 < n$, since $n+2k-1-n=3$, we know that in addition to that row 1 will become clean after the second row-column sort, three zeros will be moved up to row 2. These three extra zeros will be merged with $\gamma_3(1)$ and $\gamma_4(1)$ at row 2 and we have $2k-1+2k-5=2$ zeros left after row 2 becomes clean. In the same way, if there are kn zeros, $k>2$, at least k rows will be clean and after these k rows become clean, at least k extra zeros will be popped up to rows $k+1$ through n . This means that after the second row-column sort at least k clean rows will be generated and therefore, only $\left\lceil \frac{l_{p+k}(1)}{2} \right\rceil - k$ dirty rows will exist after the second row-column sort if there are initially $(p+k)n$

zeros in the matrix Q . Since $\frac{n}{2}+k \leq l_{p+k}(1) \leq \frac{n}{2}+2k$, $\left\lceil \frac{l_{p+k}(1)}{2} \right\rceil - k \leq \left\lceil \frac{l_p(1)}{2} \right\rceil - 0$ (0 represents that no clean row is generated), we know that if this sorting algorithm can sort an input sequence with pn zeros and a maximum of $n/2$ dirty rows, it can sort any input sequence with up to n^2 zeros.

Similarly, assume that the input sequence with qn zeros has the maximum number of $n/4$ dirty rows, i.e., $l_q(1)=n/4$. Then, $\frac{n}{4}+3k \leq l_{q+k}(1) \leq \frac{n}{4}+4k$ if there are kn more zeros in the sequence, and $\gamma_1(1) \geq \frac{n}{4}+3k$, $\gamma_2(1) \geq \frac{n}{4}+3k-1$, ..., $\gamma_t(1) \geq \frac{n}{4}+3k-t$. After the second row-column sort, $\gamma_1(1)$ and $\gamma_2(1)$ will be merged as $\gamma_1(2)$, $\gamma_3(1)$ and $\gamma_4(1)$ will be merged as $\gamma_2(2)$, and so on. Therefore, we have $\gamma_1(2) \geq \frac{n}{2}+6k-1$, $\gamma_2(2) \geq \frac{n}{2}+6k-5$, ..., $\gamma_t(2) \geq \frac{n}{2}+6k-(4t-3)$. Then, after the third row-column sort, $\gamma_1(2)$ and $\gamma_2(2)$ will be merged as $\gamma_1(3)$, $\gamma_3(2)$ and $\gamma_4(2)$ will be merged as $\gamma_2(3)$, and so on. We have $\gamma_1(3) \geq n+12k-6$, $\gamma_2(3) \geq n+12k-22$, ..., $\gamma_t(3) \geq n+12k-16t-10$. Since $n+12k-6 > n$ for $k=1$ and $n+12k-22 > n$ for $k=2$, we will have one and two clean rows, respectively, after the third row-column sort. For $k=3$, although $n+12k-38 < n$, since $n+12k-6-n=30$ and $n+12k-22-n=14$, that is, in addition to that row 1 and row 2 will become clean after the third row-column sort, at least 44 extra zeros will be popped up to row 3 and row 4. Two of these extra zeros will be merged with $\gamma_5(2)$ and $\gamma_6(2)$ at row 3 and we will have 42 zeros left after row 3 becomes clean. In the same way, if there are kn zeros, $k>3$, at least k rows will be clean and after these k rows become clean, at least 42 extra zeros will be moved up to rows $k+1$ through n . That is, after the third row-column sort, at least k clean rows will be generated and only $\left\lceil \frac{l_{q+k}(1)}{4} \right\rceil - k$ dirty rows will exist after the third row-column sort if there are initially $(p+k)n$ zeros in the matrix Q . Since $\left\lceil \frac{l_{q+k}(1)}{4} \right\rceil - k \leq \left\lceil \frac{l_q(1)}{4} \right\rceil - 0$, we know that if

this sorting algorithm can sort an input sequence with qn zeros and a maximum of $n/4$ dirty rows, it can sort any input sequence with up to n^2 zeros.

These results can be generalized to an input sequence with zn zeros and a maximum of $n/2^m$ dirty rows, i.e., $l_z(1)=n/2^m$. Then, $\frac{n}{2^m}+(2m-1)k \leq l_{z+k}(1) \leq \frac{n}{2^m}+2mk$ if there are kn more zeros in the sequence. After the $(m+1)$ th row-column sort at least k clean rows will be created, therefore, only $\left\lceil \frac{l_{z+k}(1)}{2^m} \right\rceil - k$ dirty rows will exist after the third row-column sort if there are initially $(z+k)n$ zeros in the matrix Q . Since $\left\lceil \frac{l_{z+k}(1)}{2^m} \right\rceil - k \leq \left\lceil \frac{l_z(1)}{2^m} \right\rceil - 0$, we know that if this sorting algorithm can sort an input sequence with zn zeros and a maximum of $n/2^m$ dirty rows, it can sort any input sequence with up to n^2 zeros. Thus, if the sorting algorithm can sort an input sequence with n zeros it can sort any input sequence with up to n^2 zeros. Based on this result and Theorem 4.1, we know that the trapezoid sort is a complete sorting algorithm. \square

From the analysis in Theorem 4.3, the number of iterations in step 4 of the trapezoid sort (Fig. 4.2) is determined by the maximum number of dirty rows that can be generated after step 3 (i.e., the first iteration) by an input sequence with n zeros. The following corollary follows immediately.

COROLLARY 4.1: The number of iterations l in the trapezoid sort to sort any 0/1 input sequence is equal to the maximum height of a trapezoid that can exist with n zeros in the matrix initially, i.e., $l=l_1(1)$. \square

4.4. Summary

This chapter presents an improved algorithm for sorting on two-dimensional SIMD arrays. Like the "parallel bubble sort" of Sado and Igarashi [19] and the "shear sort" of Scherson and Sen [21], the "trapezoid sort" is suboptimal for this architecture. However, the justification of the algorithm is in its data movements simplicity. The major result is a reduction in the time complexity over these schemes, by a constant factor of approximately two. For practical sized arrays, there is a significant advantage of this scheme.

An advantage of this algorithm over a straightforward mapping of bitonic sort is the reduction in the time complexity by the same constant factor. It should be noted that the complicated data movements in mapping bitonic sort arise only if $O(n)$ performance is attempted. Simple column transfers alternating with individual column sorts will achieve $O(n \log_2 n)$ performance, while preserving simplicity in data movements. However, the proposed algorithm improves over this scheme by a factor of two.

CHAPTER 5.

THE Kth SMALLEST VALUE EXTRACTION

5.1. Introduction

A new two-dimensional sorting algorithm, the *trapezoid sort* [80], has been proposed in chapter 4. It preserves the properties of simple control hardware and ease of implementation. Moreover, the complexity is improved to $\lceil \log_2 l \rceil + 1$ iterations with $l \approx \sqrt{n}$. Similar to the *parallel bubble sort* and the *shear sort*, this algorithm is based upon a repeated application of the *bubble sort* technique to the rows and columns of the array. In addition, a simple *cyclic shift* operation is incorporated into the algorithm to improve the time complexity. In this chapter, we will refer any one of the above three algorithms as the *parallel row-column sort* algorithm since they all contain the two basic operations: the row-sort and the column-sort.

In this chapter, based on the {0-1} principle, the relationship between two rows in the array will be derived in section 5.2. From the results in [81, 21, 82], the number of zeros in the j th row of the array after each iteration has been shown to be *no less than* that in the k th row, for all $1 \leq j < k \leq n$. However, we can further show that the number of zeros in the j th row of the array after each iteration is always *greater* than that in the k th row unless it is equal to 0 or n . In addition, the relationship between the numbers of maximum dirty rows generated by input sequences with arbitrary number of zeros is also derived in section 5.2. This will be used in section 5.3 to determine the minimum number of clean rows generated after each iteration and

derive a more efficient method to find the k th smallest value of the inputs. The proposed method preserves the properties of the *parallel row-column sort* such as simple control hardware and ease of implementation. In addition, it requires less time complexity than the algorithms in [19,21] and [80]. As will be derived in section 5.3, approximately $\frac{3}{4}n$ processing steps are reduced in the first column-sort after the second iteration and the remaining steps will be reduced further by half after every successive iteration. Reduction on processing steps also means reduction in silicon area when the algorithm is implemented as a VLSI sorting network.

5.2. Properties of the Trapezoid Sort

Let i represent the number of iterations of the row-column sort which has already been applied on Q , Q^i represent the array after i iterations of the row-column sort with a cyclic shift operation inserted between the row-sort and the column-sort in the first iteration which includes steps 1 to 3. Let Q' represent the array after the cyclic shift which follows the first row-sort. The configuration of Q' is shown in Table 5.1 Properties of the *trapezoid sort* will be analyzed by applying the *zero-one* ($\{0-1\}$) *principle* [5]. For completeness, the $\{0-1\}$ principle is restated in Theorem 5.1.

THEOREM 5.1: If a network with n input lines sorts all 2^n sequences of 0's and 1's into nondecreasing order, it will sort any sequence of n numbers into nondecreasing order.

For the purpose of applying the $\{0-1\}$ principle, Q is assumed to contain only 0's and 1's and a row is said to be *clean* if it contains identical elements, *i.e.*, only 0's or only 1's, otherwise it is *dirty*. Without loss of generality, in the following analysis we will consider input

Table 5.1. An example output of step 2 with $n=7$.

1	2	3	4	5	6	7
7	1	2	3	4	5	6
6	7	1	2	3	4	5
5	6	7	1	2	3	4
4	5	6	7	1	2	3
3	4	5	6	7	1	2
2	3	4	5	6	7	1

sequences with pn zeros only where p can be any integer and $p \leq n$. Since in step 5 of the *trapezoid sort* algorithm, a final row-sort is used to sort the output sequence into the *SLRM* order after all elements are in their final row positions following step 4. Therefore, if the sorting algorithm can sort an input sequence with pn zeros or $(p+1)n$ zeros, then it can sort any input sequence with the number of zeros between pn and $(p+1)n$. The following theorem shows how to find the maximum number of dirty rows in the matrix Q after the first iteration of the row-column sort.

THEOREM 5.2: Let the maximum number of dirty rows in Q^i be $l_p(i)$. If there are initially pn zeros in Q , the relationship between the maximum number of dirty rows in Q^1 and pn will follow an equation similar to that of calculating the area of a trapezoid. That is, $\frac{l_p(1) \cdot [l_p(1)+1]}{2} + r_p = pn$, or equivalently, $\frac{l_p(1) \cdot [l_p(1)+1]}{2} \leq pn < \frac{[l_p(1)+1] \cdot [l_p(1)+2]}{2}$. The number of remaining zeros which do not increase the height of the trapezoid is represented by r_p ($\leq l_p(1)$) in the above equation.

PROOF: Since the operation of the first column-sort (*i.e.*, step 3 in Fig. 4.2) is just to move zeros in each column of Q' to the top of Q^1 , the maximum number of dirty rows in Q^1

can be regarded as the maximum number of zeros allowed in any column of Q' .

To have a zero located at $Q'[l, m]$, from Table 5.1 we know that at least $n-[l-(m+1)]$ and $m-l+1$ zeros are required in row l for $l > m$ and $l \leq m$, respectively. For Q' to have the maximum number h of dirty rows with the least total number of zeros, there should be h zeros in the same column and in continuous rows, i.e., located in $Q'[l, l], Q'[l-1, l], Q'[l-2, l], \dots, Q'[l-h+1, l]$ for $l-h \geq 0$ or in $Q'[l, l], Q'[l-1, l], \dots, Q'[1, l], Q'[n, l], Q'[n-1, l], \dots, Q'[n-(h-l), l]$ for $l-h < 0$. Therefore, with a total of $1+2+\dots+h$ zeros and arranged in the matrix Q' according to the above restrictions (i.e., topologically equivalent to a trapezoid), we can have the maximum number of h dirty rows in Q' by having the least number of total $\frac{h(h+1)}{2}$ zeros.

In the case of having pn zeros in Q , h will be equal to $l_p(1)$ and $\frac{l_p(1)[l_p(1)+1]}{2} + r_p = pn < (p+1) \cdot n$. That is, $l_p^2(1) + l_p(1) - 2(p \cdot n - r_p) = 0$ and $l_p^2(1) + l_p(1) - 2[(p+1) \cdot n - r_p] < 0$. In other words, $l_p(1) = \frac{-1 + \sqrt{1 + 8(p \cdot n - r_p)}}{2}$.

□

An example with $p=1$ is shown in Table 5.2, where $n_1=7$, $l_1(1)=3$, $\gamma_1(1)=4$ and $r_1=1$. Also, from Table 5.2 we can see that with the maximum number of dirty rows in Q^1 or Q' , the number of zeros in row 1 of Q^1 will be at least as large as the maximum number of dirty rows (i.e., $\gamma_1(1) \geq l_p(1)$).

In the following analysis, the relationship between row j and row $j+1$ in Q^i will be obtained first in subsection A based on the number of zeros in Q^i . From the results in [81, 21, 82], the number of zeros in the j th row of Q^i has been shown to be *no less than* that in the k th row, for all $1 \leq j < k \leq n$. However, due to step 2 in the algorithm we can further show that

the number of zeros in the j th row of Q^i is always *greater* than that in the k th row unless the number of zeros in it is equal to 0 or n . In subsection B, the relationship between the numbers of maximum dirty rows generated by input sequences with arbitrary number of zeros is derived. In section V, this will be used to determine the minimum number of clean rows generated after each iteration of the row-column sort and to derive a more efficient method to find the k th smallest value of the inputs.

A. Relationship Between Two Rows

LEMMA 5.1: Let the number of zeros in row j after i iterations be represented by $\gamma_j(i)$. For any j and k such that $1 \leq j < k \leq n$, the number of zeros in the j th row of Q^i is no less than that in the k th row.

PROOF: If the number of zeros in row j of Q^i is less than that of row k , then there will be at least a zero in $Q^i[k, m]$ and a one in $Q^i[j, m]$, $1 \leq m \leq n$. This is in contradiction to our assumption that all columns are sorted in nondecreasing order after a column-sort. Thus, the zero in $Q^i[k, m]$ should move up to $Q^i[j, m]$. Therefore, for $1 \leq j < k \leq n$, there are at least as many zeros in the j th row as in the k th row. That is, $\gamma_j(i) \geq \gamma_k(i)$, for $1 \leq j < k \leq n$. □

From Lemma 5.1, we have $\gamma_j(i) \geq \gamma_{j+1}(i)$. For $i=1$, i.e., after the first iteration which includes a cyclic shift operation, we can further have the following lemma.

LEMMA 5.2: In Q^1 , the number of zeros in row j is greater than that of row $j+1$, unless $\gamma_j(1) = \gamma_{j+1}(1) = 0$ or n . That is, $\gamma_1(1) = \gamma_2(1) = \dots = \gamma_j(1) = n > \gamma_{j+1}(1) > \gamma_{j+2}(1) > \dots > \gamma_{j+k}(1) = \dots = \gamma_n(1) = 0$.

PROOF: For $\gamma_j(1)$ to be equal to $\gamma_{j+1}(1)$, every zero in row j of Q^1 must have a corresponding zero in the same column of row $j+1$, otherwise, $\gamma_j(1)$ will be greater than $\gamma_{j+1}(1)$. Since the operation of the first column-sort is to move zeros in each column of Q' to the top of Q^1 , this means that if a column of Q^1 has no less than j zeros, there will be at least $j+1$ zeros in that column and at least $j+1$

Table 5.2. The maximum number of dirty rows = 3 for $Q_{7 \times 7}$.

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	0	0	0	0	1
1	1	1	0	0	1	1
1	1	1	1	0	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

zeros in those columns of Q' with no less than j zeros. But this situation can exist only when all the columns in Q' have at least $j+1$ zeros or are all *empty* (the word *empty* means a column or a row with no zero in it). The reason is detailed in the following paragraphs.

After the first row-sort and the cyclic right shift, the array is represented as Q' . If there is a zero in $Q'[l, m]$, there are at least $n - [l - (m+1)]$ zeros in row l if $l > m$, and $m - l + 1$ zeros in row l , otherwise. These zeros are located at $Q'[l, l], \dots, Q'[l, n], Q'[l, 1], \dots, Q'[l, m]$, if $l > m$, and $Q'[l, l], \dots, Q'[l, m]$, otherwise. For $\gamma_1(1) = \gamma_2(1)$ to be true, every zero in row l of Q' should be *covered* (a zero is covered by another zero if for a zero in $Q'[l, m]$, there is also a zero in $Q'[d, m]$, $d \neq l$) by another zero in the corresponding column. So there will be at least another zero in column m . Let it be located at $Q'[d, m]$. We have the following three cases.

Case 1: for $m \geq l$ and $m \geq d$, (a) $l < d$, the number of zeros in $Q'[l, l], \dots, Q'[l, m]$ will be greater than that in $Q'[d, d], \dots, Q'[d, m]$ unless row d contains all zeros, because $l - m + 1 > d - m + 1$. (b) $l > d$, the number of zeros in row d will be greater than that in row l unless row l contains all zeros, because $l - m + 1 < d - m + 1$. *Case 2:* for $m < l$ and $m < d$, (a) $l < d$, the number of zeros in $Q'[l, l], \dots, Q'[l, n]$ and $Q'[l, 1], \dots, Q'[l, m]$ will be greater than that in $Q'[d, d], \dots, Q'[d, n]$ and $Q'[d, 1], \dots, Q'[d, m]$ unless row d contains all zeros, since $m - l + 1 > m - d + 1$. (b) $l > d$, the number of zeros in row d will be greater than that in row l

unless row l contains all zeros, since $m-l+1 < m-d+1$. In the same way, we can show that

Case3: for $m>l$ and $m<d$, (a) $l\neq 1$, the number of zeros in row d will be greater than that in row l . (b) $l=1$, the number of zeros in row d will be greater than that in row l . *Case4:* for $m<l$ and $m>d$, (a) $d\neq 1$, the number of zeros in row l will be greater than that in row d . (b) $d=1$, the number of zeros in row l and right of column m will be greater than that in row d .

From the above discussion, we know that if we add a zero to column m (i.e., $Q'[d, m]$) to cover the zero at $Q'[l, m]$, another column with only one zero at either row l or row d which need to be covered again is generated. Hence, columns that need to be filled with zeros will be generated recursively, unless every column in Q' has either two zeros or is *empty*. In the same way, if there are more than j zeros in column m of Q' , then there will be at least $j+1$ zeros in each column of Q' in order to have $\gamma_j(1)=\gamma_{j+1}(1)$. Therefore, we have

$$\gamma_1(1)=\gamma_2(1)=\dots=\gamma_j(1)=\gamma_{j+1}(1)=n>\gamma_{j+2}(1)>\dots>\gamma_{j+k}(1)=\dots=\gamma_n(1)=0. \quad \square$$

Two neighbor rows are sorted in opposite directions by the row-sort in steps 4 and 5 of the trapezoid sort algorithm. After the first row-sort in step 4, all zeros in an odd row are moved to the left and all zeros in an even row are moved to the right. From Lemma 5.2, we have $\gamma_j(1)>\gamma_{j+1}(1)$. In the following theorem, it is shown that $\gamma_j(1)$ and $\gamma_{j+1}(1)$ as well as $\gamma_{j+2}(1)$ and $\gamma_{j+3}(1)$ are combined together to form new rows after the second iteration of the row-column sort. Therefore, we can further show that $\gamma_j(i)+\gamma_{j+1}(i)\geq\gamma_{j+2}(i)+\gamma_{j+3}(i)$ based on the fact that $\gamma_j(1)+\gamma_{j+1}(1)\geq\gamma_{j+2}(1)+\gamma_{j+3}(1)$.

THEOREM 5.3: The number of zeros in row j of Q^i is greater than that in row $j+1$, unless $\gamma_j(i)=\gamma_{j+1}(i)=0$ or n . That is, $\gamma_1(i)=\gamma_2(i)=\dots=\gamma_j(i)=n>\gamma_{j+1}(i)>\gamma_{j+2}(i)>\dots>\gamma_{j+k}(i)=\dots=\gamma_n(i)=0$.

PROOF: After executing the second row-sort (the first row-sort in step 4), all zeros in row $2b-1$ are on the left and all zeros in row $2b$ are on the right ($1 \leq b \leq \frac{n}{2}$). Depending on the values of $\gamma_{2b-1}(1)$ and $\gamma_{2b}(1)$, sorting the columns will result in one of the two cases: (a) $\gamma_{2b-1}(1) + \gamma_{2b}(1) \leq n$ for $1 \leq b \leq \frac{n}{2}$, (b) $\gamma_{2b-1}(1) + \gamma_{2b}(1) > n$ for $1 \leq b < j < \frac{n}{2}$ and $\gamma_{2b-1}(1) + \gamma_{2b}(1) \leq n$ for $j < b \leq \frac{n}{2}$. In (a), all zeros in row $2b-1$ will move up $b-1$ rows and zeros in row $2b$ will move up b rows. Therefore, zeros in rows $2b-1$ and $2b$ will be merged at row b after the second column-sort (the first column-sort in step 4). As proved in Lemma 5.2, we have $\gamma_1(1) > \gamma_2(1) > \dots > \gamma_n(1)$ (since $\gamma_{2b-1}(1) + \gamma_{2b}(1) \leq n$ for $1 \leq b \leq \frac{n}{2}$), and therefore,

$$\gamma_1(1) + \gamma_2(1) > \gamma_3(1) + \gamma_4(1) > \dots > \gamma_{j-1}(1) + \gamma_j(1) = \dots = \gamma_{n-1}(1) + \gamma_n(1) = 0.$$

That is,

$$\gamma_1(2) > \gamma_2(2) > \dots > \gamma_{j/2}(2) = \dots = \gamma_n(2) = 0.$$

In case (b), if $j=1$ and $\gamma_1(1) + \gamma_2(1) = n + c > n$ ($c \geq 0$), there will be c overlapped zeros between row 1 and row 2 after the second row-sort. Since we assume that all columns are sorted in nondecreasing order, these overlapped zeros will be moved down to row 2 and combined with two merged rows 3 and 4 after the second column-sort. They will first be used to fill the *vacancies* of row 2 after the merging, then moved down to row 3 if there is already a zero in the same column. (The number of vacancies of a row represents the number of 1's in a dirty row since they can accept 0's moved down from the upper row.) If $j > 1$, the situation will be the same as that of *Case(b)* if $\gamma_{2b-1}(1) + \gamma_{2b}(1) > n$, and the same as that of *Case(a)* if $\gamma_{2b-1}(1) + \gamma_{2b}(1) < n$. Therefore, we have $\gamma_1(2) = \gamma_2(2) = \dots = \gamma_{j-1}(2) = n > \gamma_j(2) > \dots > \gamma_{j+k}(2) = \dots = \gamma_n(2) = 0$. Similarly, the result can be generalized to $i > 2$, i.e.,

$$\gamma_1(i) = \gamma_2(i) = \dots = \gamma_j(i) = n > \gamma_{j+1}(i) > \gamma_{j+2}(i) > \dots > \gamma_{j+k}(i) = \dots = \gamma_n(i) = 0.$$

□

B. Relationship Between the Numbers of Maximum Dirty Rows

In order to have maximum number of dirty rows, from the proof of Theorem 5.2 we know that $\gamma_1(1)=l_p(1)$ only if $r_p=0$, otherwise, $\gamma_1(1)>l_p(1)$, and $\gamma_j(1)\geq\gamma_{j+1}(1)+1$ for all $\gamma_j(1)\neq 0$ or n . From the proof of Theorem 5.3, we know that one clean row can be generated after the second row-column sort if $\gamma_1(1)+\gamma_2(1)\geq n$, after the third row-column sort if $\gamma_1(2)+\gamma_2(2)>\frac{n}{2}$, and so on. Therefore, in the following analysis, we will derive the relationship between l_p and the number of zeros in the input.

LEMMA 5.3: If $l_p(1)\geq\frac{n}{2f}$, then $p\geq\frac{n+2f}{2\times(2f)^2}$.

PROOF: Let $l_p(1)=\frac{n}{2f}$, since $l_p(1)^2+l_p(1)-2[p\cdot n-r_p]=0$ and $r_p\geq 0$, we have $(\frac{n}{2f})^2+(\frac{n}{2f})-2pn+2r_p=0$.

Therefore, $n\cdot[\frac{n}{(2f)^2}+\frac{1}{2f}-2p]<0$ ($r_p>0$), and $n\cdot[\frac{n}{(2f)^2}+\frac{1}{2f}-2p]=0$ ($r_p=0$). Since $n>0$, we have $p\geq\frac{n+2f}{2\times(2f)^2}$.

□

If $f=1$, $l_p(1)=\frac{n}{2}$ and $p\geq\frac{n+2}{8}$. Since both p and $l_p(1)$ are integers, we can assume that $l_p(1)=\left\lceil\frac{n}{2}\right\rceil$ and $p=\left\lceil\frac{n+2}{8}\right\rceil$. This means that if there are initially more than $(\frac{n+2}{8})\times n$ zeros in the $n\times n$ 0/1 input array, the maximum number of dirty rows generated after the first iteration of row-column sort will not be less than $\frac{n}{2}$ and at least one clean row will be generated at the top of Q^2 after the second iteration of row-column sort (this will be proved in Lemma 5.5).

If there are more than pn zeros in the array, in order to determine the number of extra clean rows generated, we should derive the relationship between $l_p(1)$ and $l_{p+k}(1)$ first.

LEMMA 5.4: For any $z=p+k$ and $p=\left\lceil \frac{n+2}{8} \right\rceil$, $l_z(1) \geq l_p(1)+k$ and $l_z(1) \leq l_p(1)+2k$.

PROOF: For $l_p(1)=\left\lceil \frac{n}{2} \right\rceil$, we have $p=\left\lceil \frac{n+2}{8} \right\rceil \geq \frac{n+2}{8}$ and

$$l_p(1)+k=\frac{n}{2}+k=\frac{(n+2k)}{2}=\frac{\sqrt{n^2+4kn+4k^2}}{2}.$$

For $l_z^2(1)+l_z(1)-2(p+k)n \geq 0$,

$$l_z(1) \geq \frac{-1+\sqrt{1+8(p+k) \cdot n}}{2} \geq \frac{-1+\sqrt{1+8[(n+2)/8+k] \cdot n}}{2}.$$

Hence,

$$l_z(1)-(l_p(1)+k) \geq \frac{-1+\sqrt{1+8[(n+2)/8+k] \cdot n}}{2} - \frac{\sqrt{n^2+4kn+4k^2}}{2}.$$

To prove that $l_z(1)-(l_p(1)+k) \geq 0$, we can check whether $-1+\sqrt{1+(n+2+8k)n}-\sqrt{n^2+4kn+4k^2} \geq 0$.

Because

$$1+\sqrt{n^2+4kn+4k^2}=\sqrt{[(n+2k)+1]^2}=\sqrt{n^2+4kn+2n+(2k+1)^2}$$

and

$$\sqrt{1+(n+2+8k)n}=\sqrt{n^2+4kn+2n+4kn+1},$$

if $-1+\sqrt{1+(n+2+8k)n}-\sqrt{n^2+4kn+4k^2} \geq 0$ is to be satisfied, $4kn+1$ should not be less than $(2k+1)^2$

or equivalently, $n \geq k+1$.

Since $p+k \leq n$, we have $k+1 \leq n-\left\lceil \frac{n+2}{8} \right\rceil$. This implies that $n \geq k+1$. Therefore,

$l_z(1) \geq l_p(1)+k$. In addition, $l_p(1)+2k=\frac{n}{2}+2k=\frac{(n+4k)}{2}=\frac{\sqrt{n^2+8kn+16k^2}}{2}$. Therefore,

$$\begin{aligned} l_p(1)+2k-l_z(1) &= \frac{\sqrt{n^2+8kn+16k^2}}{2} + 1 - \frac{-1+\sqrt{1+8[(n+2)/8+k] \cdot n}}{2} \\ &= \frac{\sqrt{n^2+8kn+16k^2}}{2} + 1 - \frac{\sqrt{n^2+4kn+2n+1}}{2} \end{aligned}$$

$$= \sqrt{n^2+8kn+2n+(4k+1)^2} - \sqrt{n^2+8kn+2n+1} \quad (2)$$

The only difference between the two polynomials inside the two square roots of Equation (2) is the term $(4k+1)^2$ in the first square root and the term $2n+1$ in the second square root. Since n can be viewed as a constant, the first polynomial is then an increasing function with respect to k^2 . Therefore, if the equation is greater than zero for $k=0$, then it is also greater than zero for $k>0$. For $k=0$, Equation (2) is equal to $n+1-(n+1)=0$, and for $k=1$ it is equal to $\sqrt{n^2+10n+25}-\sqrt{n^2+10n+1}$ which is greater than 0. Therefore, $l_z(1) \leq l_p(1)+2k$. □

5.3. Finding the Kth Smallest Element

In the analysis of speech data and in image processing, many linear filters have been used to enhance the data by smoothing the signal and removing noise. However, they have some disadvantages such as complicating the detection of edges and attributing some significance to widely spurious values. In recent years, median filters have been suggested in digital signal processing [83, 84] and image processing [85] as simple nonlinear filters to remove noise from the input signal without these disadvantages. In order statistic analysis or selection [86], the median value extractor is widely used in statistic analysis such as a descriptive measure of the center of a set of data or testing the randomness of samples consisting of numerical data [87].

A straightforward way to implement the median value extraction is to use a mesh-connected processor array to sort the input numbers x_i first, $i = 1, 2, \dots, N$. Then the median value is extracted from the $\frac{N+1}{2}$ th largest output if N is odd and the mean of the $\frac{N}{2}$ th and $\frac{N+2}{2}$ th largest outputs if N is even. However, from the following analysis we will show that a more efficient way can be derived to find the median value of the array without all inputs being

sorted in correct order and this result is then generalized to find the k th smallest element in an input sequence S for solving order statistic or selection problems.

A. Generating Clean Rows

To determine the number of clean rows generated at the top of Q^2 for an input array with arbitrary number of zeros, we will first find the smallest number of zeros required to generate a clean row at the top of Q^2 and then the number of extra clean rows generated if there are more zeros. For $\gamma_1(1) < \frac{n}{2}$, we know from Lemma 5.2, $\gamma_2(1)$ will be less than $\frac{n}{2} - 1$ and there will be no clean row generated after the second row-column sort since the total number of zeros is less than n , i.e., $\gamma_1(2) = \gamma_1(1) + \gamma_2(1) < n$. Therefore, to generate one clean row at the top of Q^2 , $\gamma_1(1)$ should be greater than $\frac{n}{2}$. For $\gamma_1(1) > \frac{n}{2}$, we know from Theorem 5.2 that to obtain the maximum number of dirty rows after the first row-column sort, $l_p(1)$ at least should not be less than $\frac{n}{2}$, and from Lemma 5.3 we know that more than $\frac{n+2}{8} \times n$ zeros should be in the array initially. If there are more than pn zeros in the array, in order to determine the number of extra clean rows generated, we need the following lemma.

LEMMA 5.5: If there are initially pn zeros in the array Q where p is equal to $\left\lceil \frac{n+2}{8} \right\rceil$

and $r_p > 0$, at least one clean row exists at the top of Q^2 .

PROOF: Since we have $p = \left\lceil \frac{n+2}{8} \right\rceil$, $l_p(1) = \frac{n}{2}$ and $l_p^2(1) + l_p(1) - 2(p \cdot n - r_p) = 0$. According to

Lemma 5.2,

$$\begin{aligned} \gamma_1(1) &= l_p(1) + c_1, & \gamma_2(1) &= l_p(1) - 1 + c_2, \\ \gamma_3(1) &= l_p(1) - 2 + c_3, & \dots, \\ \gamma_{h-1}(1) &= 2 + c_{h-1}, & \gamma_h(1) &= 1 + c_h \end{aligned}$$

where $h=l_p(1)=\frac{n}{2}$. The values of c_i 's depend on the value of r_p and $c_i \geq c_{i+1}$ since $\gamma_1(1) > \gamma_2(1) > \dots > \gamma_h(1)$. By assuming that $r_p=1$ (the worst case), the distribution of these pn zeros is shown in Table 5.3(a), where

$$\begin{aligned}\gamma_1(1) &= l_p(1) + 1 = \frac{n}{2} + 1, & \gamma_2(1) &= l_p(1) - 1 + 0 = \frac{n}{2} - 1, \\ \gamma_3(1) &= l_p(1) - 2 + 0 = \frac{n}{2} - 2, & \dots, & \gamma_h(1) = 1.\end{aligned}$$

After the second row-sort, all zeros in the odd rows and even rows will be packed to the left and right respectively, as shown in Table 5.4(a). The zeros in row 1 and row 2 will be merged at row 1 after the second column-sort. Therefore, $\gamma_1(2) = \gamma_1(1) + \gamma_2(1)$ or $\gamma_1(2) = \frac{n}{2} + 1 + \frac{n}{2} - 1 = n$. That is, these pn zeros can generate a clean row after the second row-column sort.

It is impossible to let $\gamma_1(2) = \gamma_1(1) + \gamma_2(1) < n$, since at least $\frac{n}{2} - 1$ zeros in column $m-1$ (assuming that the position of the rightmost zero of $\gamma_1(1)$ is in column m) should be removed from row 2 to row $\frac{n}{2}$ to make $\gamma_2(1) < \frac{n}{2} - 1$ and $\gamma_1(1) + \gamma_2(1) < n$. According to Theorem 5.3, the only way to restore these zeros is to put them in row 1 and this will cause the number of zeros in row 1 to be equal to n . The value of c_i when $r_p > 1$ will be at least equal to that when $r_p = 1$, so that more zeros should be removed from columns m and $m-1$. Hence, in the case of $r_p > 1$, $\gamma_1(1) + \gamma_2(1) > n$.

In general, if the number of dirty rows is not maximum (for example, with only d dirty rows in \mathcal{Q}^1 and $d < l_p(1)$), then those zeros in rows $d+1$ through h in Table 5.3(a) should be deleted since these rows are not dirty any more. These deleted zeros can only be restored in row 1 to row d based on the condition in Theorem 5.3. That is, the numbers of zeros in row 1,

row 2, ..., and row d will be greater than those of the case having the maximum number of dirty rows. Therefore, $\gamma_1(1) + \gamma_2(1)$ will always be greater than n . That is, if there are initially pn zeros in Q and p is equal to $\left\lceil \frac{n+2}{8} \right\rceil$, at least one clean row exists at the top of Q^2 .

□

Furthermore, if there are initially $(p+k)n$ zeros in Q , $k \neq 0$, then in the following theorem we will prove that at least the first $k+1$ rows in Q^2 are clean rows.

THEOREM 5.4: If there are initially $(p+k)n$ zeros in Q where p is equal to $\left\lceil \frac{n+2}{8} \right\rceil$ and $k > 0$, then at least $k+1$ clean rows exist at the top of Q^2 .

PROOF: Let $z=p+k$ and $k > 0$. To show that $(p+k)n$ zeros can generate at least $k+1$ clean rows, we begin with the case of $k=1$ and $r_p=1$ as shown in Table 5.3(b), and then the case of $k > 1$ as shown in Table 5.3(c). For $k=1$, $r_p=1$, there will be n more zeros to be added in Table 5.3(a). To have maximum number of dirty rows, $\frac{n}{2}$ of these n zeros should be used to fill the column m until there are $l_p(1)+1$ zeros in column m (i.e., $l_p(1)+k=l_{p+k}(1)=l_p(1)+1$) and put the remaining $\frac{n}{2}$ zeros in column $m+1$. As shown in Table 5.3(b), with $h=\frac{n}{2}+1$, we have $\gamma_1(1)=\frac{n}{2}+2, \gamma_2(1)=\frac{n}{2}+1, \dots, \gamma_{h-1}(1)=3$ and $\gamma_h(1)=1$. A " ϕ " in Table 5.3 represents a newly added zero.

After the second row-sort, zeros in odd rows will be moved to the left of the rows and zeros in even rows will be moved to the right of the rows as shown in Table 5.4(b). From this figure we also know that since there are three overlapped zeros between rows 1 and 2, therefore, there will be three extra zeros left after row 1 is fully filled in the second column-sort. These overlapped zeros in columns $\frac{n}{2}$, $\frac{n}{2}+1$, and $\frac{n}{2}+2$ will be moved up to row 2 after the second

Table 5.3. The relationship between r_{p+k} and k .

0	0	0	0	0	0				
	0	0	0	0					
		0	0	0					
			0	0					
				0					

(a) $l_p(1) = \frac{n}{2}, k=0, r_p=1.$

0	0	0	0	0	0	ϕ			
	0	0	0	0	ϕ	ϕ			
		0	0	0	ϕ	ϕ			
			0	0	ϕ	ϕ			
				0	ϕ	ϕ			
					ϕ				

(b) $l_{p+1}(1) = \frac{n}{2} + 1, k=1, r_{p+1} = \frac{n}{2}.$

0	0	0	0	0	0	ϕ	0		
	0	0	0	0	ϕ	ϕ	0		
		0	0	0	ϕ	ϕ	0		
			0	0	ϕ	ϕ	0		
				0	ϕ	ϕ	0		
					ϕ	0	0		
						0	0		
							0		

(c) $l_{p+2}(1) = \frac{n}{2} + 3, k=2, r_{p+2}=0.$

Table 5.4. The distribution of zeros in each row after the second row sort.

0	0	0	0	0	0				
						0	0	0	0
0	0	0							
								0	0
0									

(a) $k=0$

0	0	0	0	0	0	ϕ			
				ϕ	ϕ	0	0	0	0
0	0	0	ϕ	ϕ					
						ϕ	ϕ	0	0
0	ϕ	ϕ							
									ϕ

(b) $k=1$

0	0	0	0	0	0	ϕ	0		
			0	ϕ	ϕ	0	0	0	0
0	0	0	ϕ	ϕ	0				
					0	ϕ	ϕ	0	0
0	ϕ	ϕ	0						
							0	0	ϕ
0	0								
									0

(c) $k=2$

column-sort. Also, after the second column-sort zeros in rows 3 and 4 will be merged in row 2. Since the total number of zeros in rows 3 and 4 is equal to $n-1$, only one vacancy will be available to accept the overlapped zeros from rows 1 and 2. Therefore, these overlapped zeros in columns $\frac{n}{2}$ and $\frac{n}{2}+2$ will be moved up to row 3 again and the overlapped zero in column $\frac{n}{2}+1$ will fill the vacancy in row 2 after the second row-sort to create another clean row. From the above discussion, we can see that two pairs of rows in Table 5.4(b), (γ_1, γ_4) and (γ_2, γ_3) , contribute to the generation of two clean rows.

Also, from Table 5.3(b), we see that at least two zeros should be removed from row 4 to make the array Q unable to generate two clean rows after the second row-column sort. According to Theorem 5.3, those zeros in columns $\frac{n}{2}+2$ and $\frac{n}{2}+1$ of Table 5.3(b) should be removed from row 4 to row h so that $\frac{n}{2}-3$ and $\frac{n}{2}+1-3$ zeros should be removed. These removed zeros can only be restored in rows 1, 2, and 3 according to Theorem 5.3. Suppose that rows 1, 2, and 3 have a , b and c more zeros to be removed, respectively. Then, $\gamma_1+\gamma_2=n+a+b+3$. These extra $a+b+3$ zeros will be moved down to row 2 and merged with $\gamma_3(1)+\gamma_4(1)=n-1+a+b+c+3$. Therefore, row 2 will also be cleaned since $a+b+c+3>1$.

For $r_p>1$, as discussed in Lemma 5.5, the number of zeros in each row after the first row-column sort will be greater than that if $r_p=1$. Therefore, they will generate $k+1$ clean rows after the second row-column sort. In general, the number of dirty rows is less than $l_p(1)+k$ and in the same way as proved in the case of $k=0$, these general cases will also generate at least $k+1$ clean rows after the second row-column sort.

If $k=2$, as shown in Table 5.3(c), $l_{p+2}(1)=l_p(1)+2+1=l_p(1)+k+1$. This means that at least there will be one more zero in $\gamma_j(1)$, for all j such that $\gamma_j(1)>0$, in addition to what it should

have (as the analysis we did for $k=1$). That is, for $h=\frac{n}{2}+k+1$,

$$\gamma_1(1)=\frac{n}{2}+k+1+c_1, \quad \gamma_2(1)=\frac{n}{2}+k+c_2, \quad \gamma_3(1)=\frac{n}{2}+k-1+c_3,$$

$$\gamma_4(1)=\frac{n}{2}+k-2+c_4, \quad \gamma_5(1)=\frac{n}{2}+k-3+c_5, \quad \dots,$$

$$\gamma_{k+1}(1)=\frac{n}{2}+1+c_{k+1}, \quad \gamma_{k+2}(1)=\frac{n}{2}+c_{k+2}, \quad \dots,$$

$$\gamma_{2k+3}(1)=\frac{n}{2}-k-1+c_{2k+3}, \quad \gamma_h(1)=1+c_h.$$

Since $c_i \geq c_{i+1}$ and $c_i \geq 0$ (according to Theorem 5.3), we know at least $k+1$ clean rows can be generated by the pairs $(\gamma_{k+1}(1), \gamma_{k+3}(1)), \dots, (\gamma_1(1), \gamma_{2k+3}(1))$. Similarly, it is impossible that the array generates less than $k+1$ clean rows as proved for $k=1$. For $r_p > 1$ or when the number of dirty rows is not maximum, the proof is similar to that for the $k=1$ case. Therefore, at least $k+1$ clean rows will be generated after the second row-column sort. \square

B. Finding the median value

From the above derived properties of the trapezoid sort, we know that once a row is clean it will remain clean during the succeeding iterations. That is, the number of iterations of the row-column sort to sort the array depends on the number of remaining dirty rows [21]. (However, this does not mean that the clean rows can be excluded from the remaining row-column sort operations.) Therefore, the property of the *trapezoid sort* in Theorem 5.4 will be further developed to find the median value from the mesh-connected processor arrays in a fast and efficient way.

Without loss of generality, in the following analysis we assume that n is an odd number and there are initially $\left\lceil \frac{n}{2} \right\rceil \times n$ ordered 0's in Q and represented by 0_i for $1 \leq i \leq m < \left\lceil \frac{n}{2} \right\rceil \times n$ and

$m = \frac{n \times n + 1}{2}$. They are ordered as $0_i < 0_{i+1}$, so that after the process of median filtering, the largest n zeros among these 0's will be extracted and the median value among these n zeros is the median value of the n^2 input elements.

Since we have $z = \left\lceil \frac{n}{2} \right\rceil$ and $p = \left\lceil \frac{n+2}{8} \right\rceil$, according to Theorem 5.4, there will be at least $(\left\lceil \frac{n}{2} \right\rceil - \left\lceil \frac{n+2}{8} \right\rceil) + 1$ (i.e., $k_c = \left\lceil \frac{n}{2} \right\rceil - \left\lceil \frac{n+2}{8} \right\rceil$) clean rows at the top of Q^2 . Also the elements in a column are sorted in *nondecreasing order* after each column-sort and therefore, the largest n zeros in the input sequence will not be on the top k_c rows of Q^2 . Similarly, we can assume that there are initially $\left\lceil \frac{n}{2} \right\rceil \times n$ ordered 1's in Q and represented by 1_i . They are ordered as $1_i < 1_{i+1}$, so that after the process of median filtering, the smallest n ones among these 1's will be extracted and the median value among these n ones is the median value of the n^2 input elements. From the {0-1} principle, the result derived by counting the number of 0's is complementary to what is derived by counting the number of 1's. Hence, according to Theorem 5.4, there will be at least $(\left\lceil \frac{n}{2} \right\rceil - \left\lceil \frac{n+2}{8} \right\rceil) + 1$ clean rows at the bottom of Q^2 . Therefore, in the third iteration of the row-column sort, only the elements of the $n - 2k_c$ rows in the middle of Q^2 should be considered, that is, only $n - 2k_c$ steps of the odd-even transposition sort are required in the column-sort. For example, if $n=29$, we have $z=15$, $p=4$, $k_c=15-4$ and $l=7$. After the second iteration of the row-column sort, there will be 22 clean rows in Q^2 and the largest n zeros will then be located between rows 12 and 18 (i.e., $\left\lceil \frac{n+2}{8} \right\rceil - 1$ rows above and below the center row, respectively). Therefore, instead of sorting 29×29 elements, we can sort the middle 7×29 elements after the second row-column sort.

C. Finding the Kth Smallest Value

The k th smallest value among the n^2 inputs can also be obtained in a fast and efficient way similar to that of extracting the median value in subsection B. Assuming that there are initially cn zeros in Q with $(c-1)n < k \leq cn$, the k th smallest value will be among the largest n zeros. Similarly, assuming that there are initially $(n-c+1)n$ ones in Q , the k th smallest value will be among the smallest n ones. Then, according to Theorem 5.4, we can count the number of clean rows containing zeros at the top of Q^2 and the number of clean rows containing ones at the bottom of Q^2 . Again from the {0-1} principle, since the result derived by counting the number of 0's is complementary to that derived by counting the number of 1's, there will always be i and j clean rows at the top and bottom of Q^2 , respectively, with $i = c - \left\lceil \frac{n+2}{8} \right\rceil + 1$ and $j = n - c + 1 - \left\lceil \frac{n+2}{8} \right\rceil + 1$. For the same reason as discussed in subsection B, the k th smallest value will not be at the top $i-1$ clean rows or the bottom $j-1$ clean rows.

From the results in subsections B and the following analysis, a more efficient way can be obtained in finding the k th smallest value by modifying the *trapezoid sort*. This is achieved by reducing the number of steps in each column-sort after the second iteration of the row-column sort to $n - (c - \left\lceil \frac{n+2}{8} \right\rceil) - (n - c + 1 - \left\lceil \frac{n+2}{8} \right\rceil)$ steps, where $c = \left\lceil \frac{n}{2} \right\rceil$ for finding the median value in the array and $c = \left\lceil \frac{k}{n} \right\rceil$ for finding the k th smallest value. Therefore, the number of steps required by the column-sort after the second iteration to find the k th smallest value from n^2 inputs can be reduced to

$$n - (i + j) + 2 = n - (c - \left\lceil \frac{n+2}{8} \right\rceil) - (n - c + 1 - \left\lceil \frac{n+2}{8} \right\rceil) = 2 \times \left\lceil \frac{n+2}{8} \right\rceil - 1 = n - 2k_c \leq \left\lceil \frac{n}{4} \right\rceil + 2. \quad \text{That is,}$$

after the second column-sort, only the elements of the $n-2k_c$ consecutive rows (starting from row $c-(\lceil \frac{n+2}{8} \rceil -1)$ to row $c+(\lceil \frac{n+2}{8} \rceil -1)$) should be considered in the remaining row-column sort.

D. Reduction on Processing Steps

In the following we will compare the result with the *parallel bubble sort* and the *shear sort* first, and then discuss further reduction on processing steps. Although no discussion was made in [19,21] for finding the k th smallest value, based on the properties of their sorting algorithms, we know that there will be $\lfloor \frac{n}{2} \rfloor$ clean rows after the first iteration of the row-sort and the column-sort and only $\lceil \frac{n}{4} \rceil$ rows will be dirty after the second iteration [21]. That is after the second iteration, each element in the array will be at most $\lceil \frac{n}{4} \rceil -1$ rows away from its final row position [81]. However, the median value in the array can be either $\lceil \frac{n}{4} \rceil -1$ rows above or below its final position. The $\lceil \frac{n}{4} \rceil -1$ rows above and below the center row in the array should all then be considered in finding the median value after the second iteration. Therefore, instead of having $2 \times \lceil \frac{n+2}{8} \rceil -1$ rows left to be processed after the second iteration as in the *trapezoid sort*, $2 \times \lceil \frac{n}{4} \rceil -1$ rows remain to be processed after the second iteration in the *shear sort* or the *parallel bubble sort*.

Further reduction on processing steps can be achieved for each column-sort after the second row-column sort. In the proof of Theorem 5.4, it is shown that every two neighbor rows

are merged to form a new clean row so that only half of the initial dirty rows will remain after each row-column sort. This means that the number of rows needs to be processed are reduced by half after each iteration. That is, half of the $\left\lceil \frac{n+2}{8} \right\rceil - 1$ rows above and below the c -th row can be discarded in the fourth row-column sort. Let $d = \left\lceil \frac{n+2}{8} \right\rceil - 1$, then only rows from $c - \left\lceil \frac{d}{2} \right\rceil$ to $c + \left\lceil \frac{d}{2} \right\rceil$ should be processed in the fourth row-column sort. In general, only rows from $c - \left\lceil \frac{d}{2^{i-3}} \right\rceil$ to $c + \left\lceil \frac{d}{2^{i-3}} \right\rceil$ should be processed in the i th row-column sort. However, in the original *trapezoid sort*, after $i = 1 + \log_2 l$ iterations, all elements in the array are in its final row position [80]. Therefore, only the c -th row should be considered in the final row-sort in order to find the k th smallest value. For example, if $n=29$, $k_c=15-4$ and $l=7$. Based on the above analysis, only rows between 12 and 18 are processed in the third row-column sort and rows between 13 and 17 are processed in the fourth iteration. Since $1 + \log_2 l = 4$, only the fifth row should be processed in the final row-sort in order to find the median value.

Reduction on processing steps also means reduction in silicon area when the algorithm is implemented as a VLSI sorting network. The sorting network will have $1 + \left\lceil \log_2 l \right\rceil$ stages where in each stage an iteration of the row-column sort is included except that a cyclic shift operation is added between row-sort and column-sort in the first stage. Originally, for sorting $n \times n$ inputs, there will be n submodules to sort n rows independently and another n submodules to sort n columns. Each submodule is an odd-even transposition sorter which includes n steps with $\frac{n-1}{2}$ sorting elements per step. However, according to the above analysis, about $\frac{3}{4}n \times n$ data elements are eliminated from the sorting process after the second row-column sort

and the remaining data elements will be reduced by half after every successive iteration. Therefore, instead of using n submodules in the third row-sort, $\frac{n}{4}$ submodules are sufficient and the number of steps in each submodule required in the third column-column sort can be reduced from n to $\frac{n}{4}$ and then half of the submodules can be reduced from every successive row-sort and half of the steps in each submodule can be reduced from every successive column-sort.

5.4. Summary

We have derived several properties for the *trapezoid sort*. The relationship between two rows in the array after each iteration of the *trapezoid sort* are derived first based on the {0-1} principle. The number of zeros in the j th row of the array after each iteration which was shown in [21,82] to be *no less than* that in the k th row, has been further shown to be always *greater* than that in the k th row unless it is equal to 0 or n , for all $1 \leq j < k \leq n$. The relationship between the numbers of maximum dirty rows generated by input sequences with arbitrary number of zeros is also obtained. This result is then used to derive a more efficient method to find the k th-smallest value mesh-connected processor arrays. The proposed method not only preserves the properties of the *row-column sort* such as simple control hardware and ease of implementation but also has less time complexity that approximately $\frac{3}{4}n$ processing steps are reduced in the first column-sort after the second iteration and the remaining steps will be reduced further by half after every successive iteration.

CHAPTER 6.

THE MODIFIED ODD-EVEN MERGE PROCEDURE

6.1. Introduction

The row-column sort algorithms on mesh-connected processor arrays, such as the *parallel bubble sort* and the *shear sort*, have the properties of very simple control hardware and ease of implementation. However, these row-column sort algorithms are based on the odd-even transposition sort such that half of the processors are idle during each basic comparison-interchange step. In addition, they are designed to sort N inputs only, where N is the number of processors in the array. If the number of elements to be sorted is larger than N , the row-column sort algorithm can not be applied directly. To overcome this, the method in [14,21] uses the merge-split operation to replace the compare-interchange (or compare-and-swap in this chapter) operation and $O(m \log_2 m) T_N$ time complexity is required to sort mN inputs where T_N represents the time complexity to sort N inputs. Although that method is simple, it is not efficient. We will show that instead of requiring $2 \times T_N$ steps to sort $2N$ inputs by the merge-split method, only $T_N + n + 3$ steps are sufficient by the proposed *merge sort* algorithm. An $O(\log_2 m)$ order of improvement is achieved by further generalizing the *merge sort* to sort mN inputs with $O(\frac{m}{2}) T_N$ time complexity. A novel *modified odd-even merge* method is proposed here which can merge m sorted sets in $O(\frac{m}{2} \cdot \log_2 m) n$ time complexity. The other advantage of the proposed method is that it is quite simple and regular. Each processor only needs to communicate with its nearest neighbor processors and concurrent data movements are restricted in a single row (column) within a time period and hence, simplifies the control structure. Therefore, it is

very suitable for sorting more than two sets of data inputs in a mesh-connected processor array. Details of the *merge sort* and *modified odd-even merge* algorithms are in section 6.2. Analysis of the time complexity will be performed in section 6.3.

6.2. The Modified Odd-Even Merge

The row-column sort algorithm can only handle N input elements which is equal to the number of processors. If the number of elements to be sorted is larger than N , the row-column sort algorithm can not be applied directly. To overcome this, the method in [14,21] distributes the elements evenly among the processors and apply the merge-split operation instead of the compare-interchange (or compare-and-swap in this chapter) operation. The "merge-split" operation is described as follows. First, processor P_1 sends its largest element to P_2 and P_2 sends its smallest element to P_1 . Then this process repeats until the largest element in P_1 is not greater than the smallest element in P_2 [35]. For example, if there are only two elements in each processor, this process can be implemented by the following substeps: (1) sort the elements in each processor, (2) route the largest element in P_1 to its neighbor processor P_2 , and P_2 routes its smallest element to P_1 , (3) sort the elements in each processor, (4) route the largest element in P_1 to P_2 and P_2 routes its smallest element to P_1 . As discussed in chapter 4, a compare-and-swap operation on two data elements in adjacent processors can be implemented by the following sequence: route left, compare, and route right. The time for a compare-and-swap is $t_{cs}=2t_r+t_c$ where t_r is the time to route and t_c is the time to compare. Therefore, $2t_{cs}$ is required to execute a merge-split operation if there are two elements in each processor. If there are m elements in each processor, $O(m\log_2 m)$ compare-and-swap steps are required if an optimal sequential sorting algorithm is used in substeps (1) and (3) of the "merge-split" opera-

tion. In the following analysis, we will show that instead of having twice the time complexity to merge two sets of N sorted inputs by the merge-split operation, only $n+3$ extra steps are sufficient to merge two sets of N sorted inputs by the proposed *merge sort*.

As discussed in chapter 4, the row-column sort type of algorithms are implemented by applying the row-sort and the column-sort repetitively. In each row-sort (column-sort), n steps of the odd-even transposition sort are executed to sort elements in the same row (column). The advantages of the row-column sort algorithms are the simple data routing required and concurrent data movements allowed only in the same direction which simplifies the control structures. However, for sorting algorithms on mesh-connected processor arrays [14, 19, 21], during each compare-swap operation half of the processors are idle as shown in Fig. 6.1. This inefficiency can be improved based on the fact that all the elements move in the same direction at a time and processors are idle in an alternating manner, *i.e.*, in odd (even) steps of the row-sort or column-sort, all processors in the even (odd) rows or columns are idle. With some modification on the substeps of a compare-swap operation, the idle processors can sort another

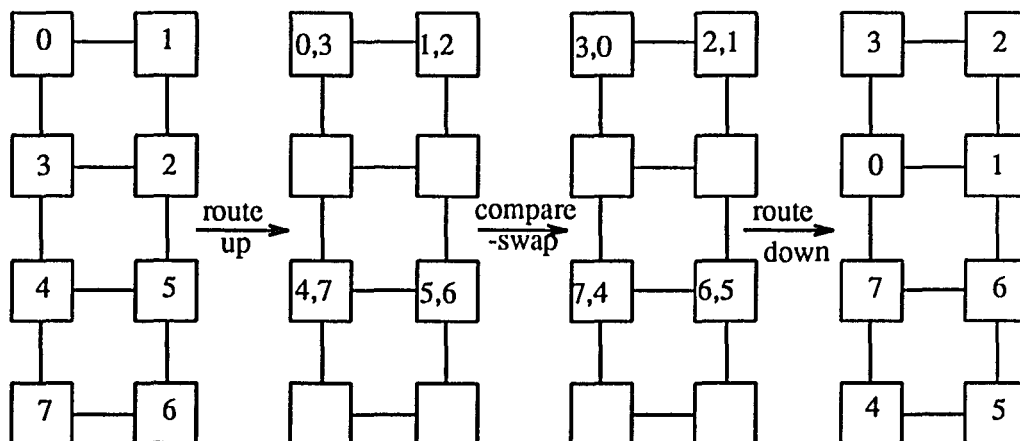


Figure 6.1. A compare-and-swap step in the column-sort.

set of input data at the same time. That is, assuming that there are two registers, R_A and R_B , in each processor and two sets of input data, N_1 and N_2 , are preloaded in the array, then instead of routing right and left in the row-sort (or up and down in the column-sort), an exchange (or swap) operation is performed between two neighbor processors and this is referred as the *modified compare-and-swap* operation. As shown in Fig. 6.2, in every data routing substep of a modified compare-and-swap step, the content in R_B of the upper (left) processor is exchanged with the content in R_A of the lower (right) processor in the column (row) sort and all processors execute the same instruction at the same time. That is, in a row-sort (column-sort) operation, if processors in the odd-numbered rows (columns) are processing N_1 , processors in the even-numbered rows (columns) are processing N_2 at the same time.

Therefore, at the time when the first input data set N_1 is being sorted and stored in R_A registers of the processors in snake-like row major order by using the *trapezoid sort*, the second input data set N_2 is also being sorted into snake-like row major order but stored in R_B registers of the processors. These two sorted data set can than be merged together into a sorted output sequence of $2N$ elements with only $n+3$ extra steps based on the *merge sort* method which will be described in subsection A. If there are more than $2N$ elements in the input array, the *merge sort* will be generalized in subsection B to sort mN inputs with $O(\frac{m}{2}) \cdot T_N$ time complexity, where T_N is the time to sort N inputs. Compared with the *merge-split* operation, an $O(\log_2 m)$ order of improvement in time complexity is achieved.

A. Sorting $2N$ Inputs

Let the two sorted sequences N_1 and N_2 be stored in R_A registers and R_B registers, respectively, of the processors in snake-like row major order. Let the processor in the (i, j) position of the array be represented by $PE_{i,j}$, $1 \leq i, j \leq n$. There are three steps in the process. In

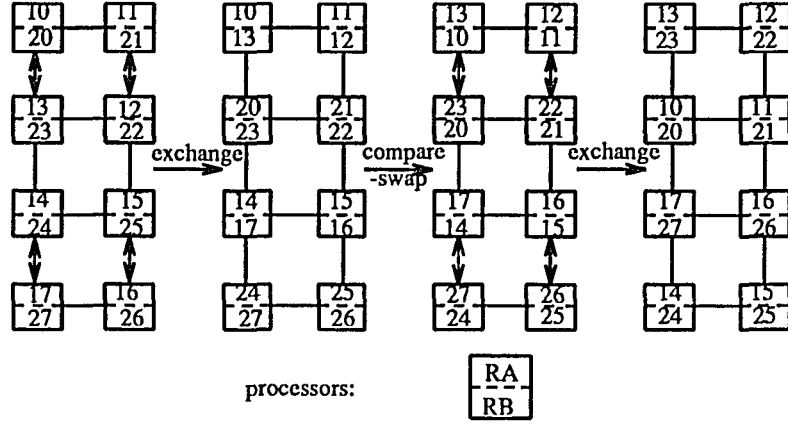


Figure 6.2. A modified compare-and-swap step in the column-sort.

the first step, the "interleaving" of the odd-even merge is executed as shown in Fig. 6.3(a). If i is odd, the content in R_B of $PE_{i,j}$ is swapped with the content in R_A of $PE_{i,j+1}$, the content in R_B of $PE_{i,j+2}$ is swapped with the content in R_A of $PE_{i,j+3}$, and there is no swapping between $PE_{i,j+1}$ and $PE_{i,j+2}$, for all odd j . If i is even, the content in R_A of $PE_{i,j}$ is swapped with the content in R_B of $PE_{i,j+1}$, the content in R_A of $PE_{i,j+2}$ is swapped with the content in R_B of $PE_{i,j+3}$, and there is no swapping between $PE_{i,j+1}$ and $PE_{i,j+2}$, for all odd j . The configuration after the interleaving process is shown in Fig. 6.3(b), where the two interleaved sequences are stored in R_A registers and R_B registers of the processors, respectively. Let these two interleaved sequences be represented by L_A and L_B .

In the second step, "sorting the two interleaved sequences" is performed. This scenario can be viewed as that the two random input sequences, L_A and L_B , are preloaded in the array. In this case, an efficient method to sort these two interleaved sequences is to sort these two sequences concurrently by the *trapezoid sort* which will require $2 \times (\lceil \log_2 l \rceil + 1)n + n$ compare-and-swap steps to complete the sorting. However, these two sequences L_A and L_B are not random input sequences, since they have already been sorted in some order. For example, for the sequence L_A , if i is odd, the content in R_A of $PE_{i,j}$ is no

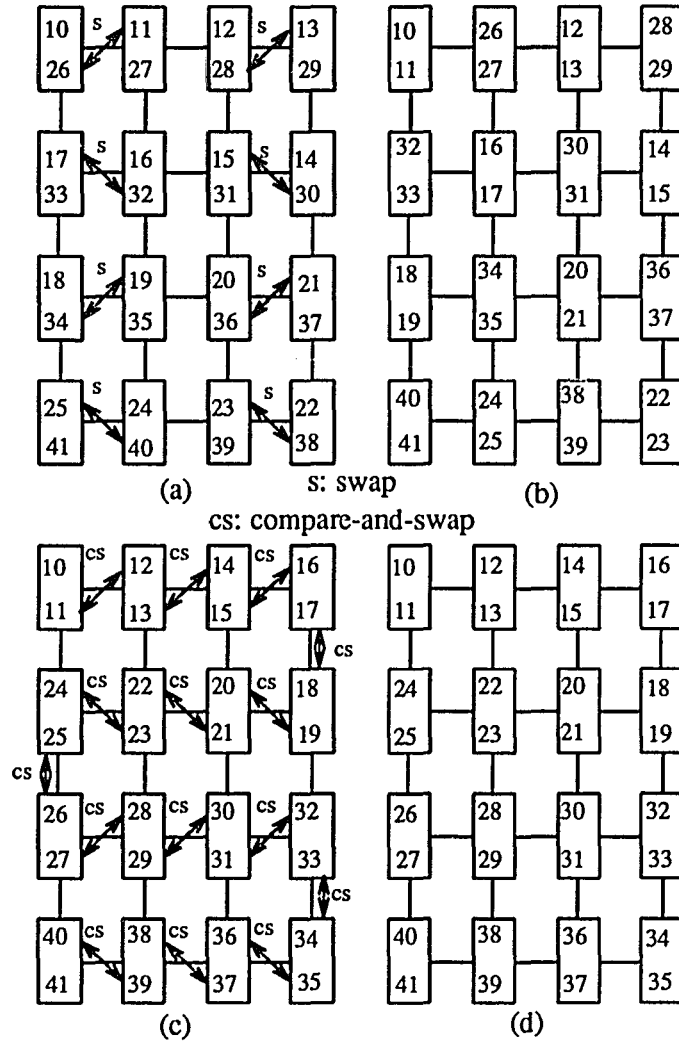


Figure 6.3. An example of the modified odd-even merge.

greater than the content in R_A of $PE_{i,j+2}$, and if i is even, the content in R_A of $PE_{i,j}$ is no less than the content in R_A of $PE_{i,j+2}$. Furthermore, L_A (or L_B) itself is generated by two shuffled sorted sequences. Let the two sorted sequences which form L_A be represented as M_1 and M_2 . Using the serial bubble sort, the worst case scenario to sort L_A is that the largest element in M_1 is less than the smallest element in M_2 , since M_1 and M_2 are shuffled to form L_A and this case has the maximum distance ($\frac{n}{2}$) to move an element to its final position. This is also the worst case in the row-column sort which is implemented

based on the odd-even transposition sort. The detail reasoning is in the following.

If the largest element in M_1 is less than the smallest element in M_2 , then M_1 can be assumed to have all zeros and the largest element of M_1 will be in $PE_{n,n-1}$ and M_2 can be assumed to have all ones and the smallest element of M_2 will be in $PE_{1,2}$. In this situation, instead of using $\lceil \log_2 l \rceil + 1$ iterations of row-column sort followed by a row-sort, a single column-sort followed by a single row-sort is sufficient to sort the sequence. We designate the operation of a single column-sort followed by a row-sort as a *single-column-row sort*. An example worst case of an 8×8 L_A is shown in Fig. 6.4. Any change of an element from 1 to 0 in make L_A such that it is no longer a worst case should be performed at $PE_{1,2}$ first. Similarly, any change of an element from 0 to 1 should be done at $PE_{n,n-1}$ first, since M_1 and M_2 are two sorted sequences. As shown in Fig. 6.4, any change from 0 to 1 or 1 to 0 will not increase the number of steps required to sort L_A , that is, $2n$ steps of the odd-even transposition sort are sufficient to sort the sequence L_A . These $2n$ steps include n steps of compare-and-swap in the column-sort and n steps of compare-and-swap in the row-sort. However, based on the fact that M_1 and M_2 are interleaved to form L_A , $\frac{n}{2}$ steps are sufficient to clean a column in the column-sort since the maximum distance required for any 0 or 1 to move to its final destination is $\frac{n}{2} - 1$. Similarly, $\frac{n}{2}$ steps in the following row-sort can clean every row after the column sort. We call this operation the

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

Figure 6.4. An example 8×8 L_A .

reduced single-column-row sort to represent the fact that instead of using n steps of the odd-even transposition sort for each of the column-sort and the following row-sort, only $\frac{n}{2}$ steps are sufficient for each.

In the third step, "merging of the two sorted sequences" is executed. This process is implemented by two compare-and-swap steps as shown in Fig. 6.3(c). Since concurrent data movements are allowed in the same direction only, the compare-and-swap step that compares two neighbor processors in the same column can not be executed until the two neighbor processors in the same row finish sorting. It should be noted that in the original odd-even merge method, another interleaving step is required between step 2 and step 3. However, for the current application, this step can be combined with the merging operation and implemented directly by comparing R_B of $PE_{i,j}$ with R_A of $PE_{i,j+1}$ (Fig. 6.3(c)). In the following, we will refer steps 1 to 3 as the *modified odd-even merge* procedure and the procedure *merge sort* describes the process of sorting the two sets of inputs at the same time first and then, performing the *modified odd-even merge*.

Therefore, with one step to perform the interleaving operation, $\frac{n}{2} + \frac{n}{2}$ steps to sort the interleaved sequence, and two more steps to complete the merging, the two sorted sequences N_1 and N_2 can be merged as a sorted $2N$ -output sequence. Thus, instead of using $2 \times [2 \times (\lceil \log_2 l \rceil + 1)n + n]$ compare-and-swap steps to sort the $2N$ -input sequence based on the trapezoid sort and the merge-split operation, only $[2 \times (\lceil \log_2 l \rceil + 1)n + n] + n + 3$ steps are required by the *merge sort*.

B. Sorting mN Inputs

In this subsection, the *merge sort* procedure will be further generalized to m input sequences where $m > 2$ and each sequence has N elements. In order to simplify the analysis, we will assume that (1) instead of only two registers R_A and R_B as in subsection A, there are m registers, R_1, R_2, \dots, R_m , in each processor (or each processor has a local memory that can store m elements) and m is a power of two, (2) each processor can access its registers (or memory) with the same speed, and (3) the mN inputs are equally distributed among processors.

Let an input sequence S with mN elements be represented as m N -element sequences. The first sequence stored in R_1 registers of the processors is represented as Q_1 , the second sequence stored in R_2 registers is represented as Q_2 , ..., and the m th sequence stored in R_m registers is represented as Q_m . From the last subsection, we know that two sets of inputs can be sorted concurrently and then merged together to form a sorted $2N$ -element sequence. Therefore, there will be $\frac{m}{2}$ sorted $2N$ -element sequences generated after the first merge. These $\frac{m}{2}$ sequences can be merged again based on the modified odd-even merge to form $\frac{m}{4}$ sorted $4N$ -element sequences, and so on until the sequence S is sorted. That is, the *modified odd-even merge* is applied recursively to merge the m sorted sequences two at a time until all of them are merged.

The generalized *merge sort* and *modified odd-even merge* procedures are described in Fig. 6.5 and Fig. 6.6, respectively. Let $|S|$ represent the number of elements in the input sequence S and it is equal to mN . (If $|S| < mN$, some values larger (smaller) than the largest (smallest) entry in the array can be used to fill S .) If there are more than $2N$ input elements in $|S|$, i.e., $|S| > 2N$, the sequence S will be equally divided into two subsequences S_1 and S_2 . The subsequence S_1 which includes Q_1 to $Q_{\frac{m}{2}}$ will be sorted first by recursively calling the

```

Procedure Merge Sort ( $S$ );
  begin
    /* divide  $S$  into two subsequences,  $S_1$  and  $S_2$ , of equal sizes */
     $S_1$  = contents in  $R_1$  registers through  $R_{m/2}$  registers of all processors; /*  $m = |S|/N$  */
     $S_2$  = contents in  $R_{m/2+1}$  registers through  $R_m$  registers of all processors;
    if  $|S| > 2N$  then
      begin
        /* sort the two subsequences recursively one after another */
        Merge Sort ( $S_1$ );
        Merge Sort ( $S_2$ );
        /* merge the two sorted subsequences  $S_1$  and  $S_2$  into a sorted sequence  $S$  */
        Modified Odd-Even Merge ( $S$ );

      end
    else
      begin
        do in parallel
          begin
            Row-Column Sort ( $S_1$ );
            Row-Column Sort ( $S_2$ );
          end
        /* merge the two sorted subsequences  $S_1$  and  $S_2$  into a sorted sequence  $S$  */
        Modified Odd-Even Merge ( $S$ );
      end
    end
  end

```

Figure 6.5. Merge sort procedure.

procedure and then followed by sorting the subsequence S_2 which includes $Q_{\frac{m}{2}+1}$ to Q_m . After the two subsequences are sorted, the procedure *modified odd-even merge* in Fig. 6.6 is used to merge these two sorted subsequences.

If there are only $2N$ inputs, as described in subsection A, the procedure will completely sort these $2N$ inputs. Since two sets of inputs can be sorted concurrently, two *Row-Column Sort* processes are executed in parallel. The *Row-Column Sort* procedure can be

implemented by any of the row-column sort algorithms, and the *trapezoid sort* is used here since it requires the least number of compare-and-swap steps. After each set of the data inputs are sorted, the procedure *modified odd-even merge* again is performed to merge the two sorted sets.

The *modified odd-even merge* procedure first interleaves (or shuffles) the two sorted sequences to be merged. When implemented in a mesh-connected array, this step means that the contents in $R_{m/4+1}$ to $R_{m/2}$ are exchanged with the contents in $R_{m/2+1}$ to $R_{3m/4}$. Thus, the subsequence formed by Q_1 to $Q_{\frac{m}{2}}$ can be further decomposed into two sorted subsequences, Q_1 to $Q_{\frac{m}{4}}$ and $Q_{\frac{m}{4}+1}$ to $Q_{\frac{m}{2}}$. Therefore, the *modified odd-even merge* can be executed again to sort the two interleaved sequences in Q_1 to $Q_{\frac{m}{2}}$ and $Q_{\frac{m}{2}+1}$ to Q_m , separately. The function *Reduced Single-Column-Row Sort* in Fig. 6.6, as described in subsection A, is implemented by a column-sort with only $\frac{n}{2}$ steps followed by a row-sort with the same number of steps.

If $m=2$, the two sorted sequences to be merged are interleaved first, as shown in Fig. 6.3(a), and then follow the steps described in subsection A to merge them as a sorted $2N$ -output sequence. An example of sorting $4N$ inputs is shown in Fig. 6.7. The random input sequence with $4N$ inputs are sorted two subsequences at a time.

At the beginning, the two subsequences in R_1 registers and R_2 registers of all processors, respectively, are sorted concurrently and merged into a sorted $2N$ -output sequence. Then the next two subsequences in R_3 registers and R_4 registers are processed. Two sorted sequences, S_1 and S_2 , with $2N$ elements each are stored in the processor array as shown in Fig. 6.7(a). The sequence S_1 consisting of 10, 11, 12, ..., 40, 41 is stored in R_1 and R_2 registers of all processors and ordered by the *merge sort* in *snake-like row major ordering*. In the same way,

Procedure *Modified Odd-Even Merge* (S);

```

begin
  /* divide  $S$  into two subsequences,  $S_1$  and  $S_2$ , of equal sizes */
   $S_1$  = contents in  $R_1$  registers through  $R_{m/2}$  registers of all processors; /*  $m = |S|/N$  */
   $S_2$  = contents in  $R_{m/2+1}$  registers through  $R_m$  registers of all processors;
  if  $|S| > 2N$  then
    begin
      interleave  $S_1$  and  $S_2$ ; /* as shown in Fig. 6.7(b) */
      Modified Odd-Even Merge ( $S_1$ );
      Modified Odd-Even Merge ( $S_2$ );
      merge the two sorted sequence  $S_1$  and  $S_2$  into  $S$ ; /* as shown in Fig. 6.7(d) */
    end
  else
    begin
      interleave the two sorted sequence  $S_1$  and  $S_2$ ; /* as shown in Fig. 6.3(a) */
      do in parallel
        begin
          Reduced Single-Column-Row Sort ( $S_1$ );
          Reduced Single-Column-Row Sort ( $S_2$ );
        end
      merge the two sorted sequence  $S_1$  and  $S_2$  into  $S$ ; /* as shown in Fig. 6.3 (c) */
    end
  end
end

```

Figure 6.6. Modified odd-even merge procedure.

the sequence S_2 with 42, 43, ..., 73 is stored in R_3 and R_4 registers.

In the second step, "interleaving" (or shuffling) of the two sequences is performed as shown in Fig. 6.7(b). By exchanging the contents in each pair of R_2 and R_3 , the two sequences in R_1 registers and R_2 registers can be viewed as two sorted sequences and the combination of these two sequences is an interleaved sequence. An example of this interleaved sequence is shown in Fig. 6.7(b) as 10, 42, 12, 44, ..., 38, 70, 40, 72.

In the third step, the *modified odd-even merge* procedure merges $2N$ elements. The contents in R_1 registers and R_2 registers are interleaved again as the process in Fig. 6.3(a). Then the contents in R_1 registers and R_2 registers are sorted by the modified column-row sort concurrently and rearranged in snake-like row major order as the process in Fig. 6.3(c). Let these two sorted sequences be represented as L_1 and L_2 . L_1 and L_2 are then merged to form a sorted $2N$ -output sequence. The ordered $2N$ -output sequence, 10, 12, 14, ..., 70, 72, is stored in the R_1 and R_2 registers of the processors. After L_1 and L_2 have been merged, the same sorting and merging process can be repeated on contents in R_3 and R_4 registers. Therefore, two sorted sequences, 10, 12, ..., 72 and 11, 13, ..., 73 are stored in the array as shown in Fig. 6.7(c). Finally, merging of these two sorted sequences is done by comparing (and exchange if necessary) the contents in each pair of R_2 and R_3 with no interleaving required before merging.

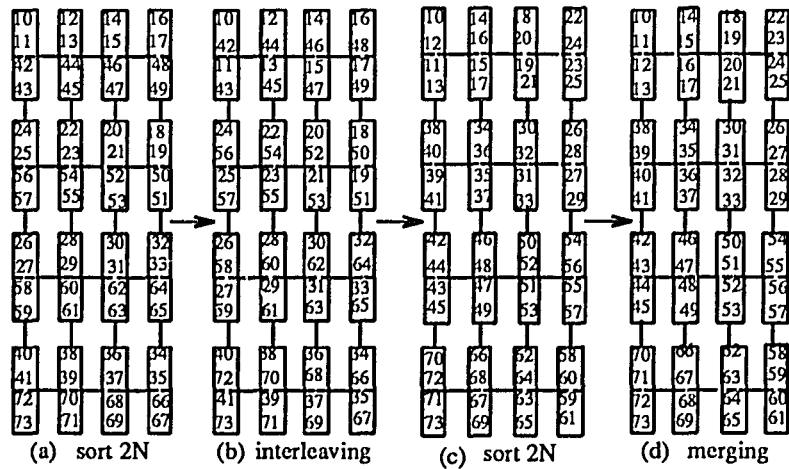


Figure 6.7. An example of sorting $4N$ inputs.

6.3. Analysis

Let T_{mN} represent the number of modified compare-and-swap steps required to sort mN inputs, *i.e.*, the time complexity of the procedure *merge sort*, and C_{mN} represent the time complexity of the procedure *modified odd-even merge*. Then, from Fig. 6.5 and Fig. 6.6, we have

$$T_{mN} = 2T_{\frac{m}{2}N} + C_{mN} \quad (6.1)$$

and

$$C_{mN} = \frac{m}{4} + 2C_{\frac{m}{2}N} + \frac{m}{4} \quad (6.2)$$

which implies

$$T_{mN} = 2(T_{\frac{m}{2}N} + C_{\frac{m}{2}N}) + \frac{m}{2}. \quad (6.3)$$

The first $\frac{m}{4}$ in (2) represents the number of exchange steps (it should be noted that the amount of time required by an exchange step is less than that of a compare-and-swap step) required to exchange the contents in $R_{m/4+1}$ through $R_{m/2}$ with the contents in $R_{m/2+1}$ through $R_{3m/4}$, respectively. An example is shown in Fig. 6.7(b). Since $m=4$, one step is required to exchange the content in R_2 with that in R_3 . The second $\frac{m}{4}$ in (2) represents the number of compare-and-swap steps required in the last merging step (as shown in Fig. 6.7(d)) to compare and exchange the contents in $R_{m/4+1}$ through $R_{m/2}$ with the contents in $R_{m/2+1}$ through $R_{3m/4}$, respectively. If there are only $2N$ inputs, from section 6.2 we know that two sets of data can be processed concurrently by the mesh-connected processors array and therefore,

$$T_N = 2(\lceil \log_2 l \rceil + 1)n + n \text{ and } C_N = \frac{n}{2} + \frac{n}{2},$$

$$T_{2N} = T_N + C_{2N} \text{ and } C_{2N} = C_N + 3$$

This implies that

$$T_{mN} \approx 2(T \frac{m}{2} N + C \frac{m}{2}) + \frac{m}{2} \approx \frac{m}{2} \times [T_{2N} + (\log_2 m - 1) \cdot C_{2N}] + \frac{m}{2} + m \cdot (\log_2 m - 2). \quad (6.4)$$

When n is large and $m \ll n$, we have

$$T_N \gg \log_2 m \cdot C_N \gg m \cdot \log_2 m$$

since $2(\lceil \log_2 l \rceil + 1)n + n \gg n \cdot \log_2 m$. Also we know that $T_{2N} \approx T_N + C_N$ and therefore,

$$T_{mN} \approx 2(T \frac{m}{2} N + C \frac{m}{2}) \approx \frac{m}{2} \times [T_{2N} + (\log_2 m - 1) \cdot C_{2N}] \approx \frac{m}{2} \times [T_N + (\log_2 m - 1) \cdot C_N]. \quad (6.5)$$

That is, instead of an $O(m \log_2 m) \cdot T_N$ time complexity to sort mN data inputs with the "merge-split" operation, only $O(\frac{m}{2}) \cdot (T_N + n \cdot \log_2 m)$ ($T_N \gg n \cdot \log_2 m$) time complexity is sufficient to sort mN data inputs by the *modified odd-even merge* procedure. Therefore, we have achieved an $O(\log_2 m)$ order improvement.

The proposed method can be modified by replacing the *Row-Column Sort* procedure in Fig. 6.5 with any sorting algorithm for the mesh-connected processors array. However, the data movements in these algorithms are not as simple as those in the class of row-column sort algorithms and thus, complex control schemes are required in these algorithms to synchronize the data movements in order to sort two sets of data inputs. To reduce the complexity of the control scheme, the "do in parallel" operation in Fig. 6.5 can be replaced by two consecutive operations and thus, $T_{2N} = 2T_N + C_{2N}$ and $C_{2N} = C_N + 3$. Therefore, the time complexity required to sort m sets of N data inputs is

$$T_{mN} \approx 2(T \frac{m}{2} N + C \frac{m}{2}) \approx m \times T_N + \frac{m}{2} \times C_{2N}, \quad (6.6)$$

which is about twice the time complexity when two sets of data inputs are sorted concurrently.

6.4. Summary

A novel *merge sort* method for the mesh-connected processor arrays is presented in this chapter. Instead of an $O(m \log_2 m) T_N$ time complexity to sort mN input elements by the previous merge-split operation, only $O(\frac{m}{2}) T_N$ time complexity is sufficient to sort mN random inputs where T_N is the time complexity of the row-column sort algorithm. Therefore, we have achieved an $O(\log_2 m)$ order of improvement in time complexity to sort mN inputs. Other advantages of the proposed method include the simplicity of the architecture and efficient data movements with only near-neighbor communications. Therefore, it is very suitable for sorting more than two sets of input elements in mesh-connected processor arrays.

CHAPTER 7.

CONCLUSIONS

7.1. Summary and Discussions

Many applications in real-time digital signal and image processing need a high performance parallel computer. Parallel sorting algorithms for two-dimensional mesh-connected processor arrays including efficient implementations of these sorting algorithms in two-dimensional VLSI models have been intensively studied. Due to the availability of VLSI and WSI technologies at a reasonable cost, use of special purpose architecture for parallel sorting on a huge amount of input data has become attractive recently.

In chapter 2, I have presented a highly reliable sorting array. It can detect multiple errors and correct a single error for on-line applications. As a systolic sorting array based on the odd-even transposition sort, it has a regular structure and simple interconnection links. Both the regularity and the simplicity of the odd-even transposition sorting array are preserved by the presented fault tolerance technique so that redundancy can fit into the system nicely, either to enhance the system performance or to replace the faulty elements. In addition, it can be reconfigured easily to tolerate the faulty sorting elements located by the on-line fault diagnosis procedure and can be degraded gracefully after redundancy is exhausted. Also, I discovered an important robust property of the odd-even transposition sorting array in which a single error can be masked automatically and multiple errors can be detected concurrently without disturbing the normal circuit operation. Therefore, extra cost incurred by bringing in fault tolerance features

can be minimized by exploiting the inherent properties of the embedded sorting algorithm. According to the analysis in section 2.6, hardware overhead for fault tolerance is about $(54+c)/14N$ and only 3 clocks delay is incurred in the pipeline. Since the sorting array is two-level pipelined and all the checkers are implemented to be fault secure or totally self-checking, it is well applicable to real-time applications which require high throughput as well as high reliability. The error detection techniques in this dissertation can be applied to sorting arrays based on other sorting algorithms with either two-level pipelined or bit-level serial structure.

Due to the large area and the processing technology limitation, defects seems unavoidable in WSI implementation. Therefore, the networks need to have defect tolerance capabilities. A novel hierarchical modular sorting network (*HMSN*) is presented in chapter 3. Since it is a comprise between the simple communication scheme of the odd-even transposition sort and the fast convergent speed of the bitonic sort, it has a good *area-time* performance. It uses less hardware and converges faster than a single-level odd-even transposition sorter and the wire complexity problem of the bitonic sorter in VLSI or WSI is alleviated. Networks with regular interconnections have been shown to be equivalent to the bitonic network and used to replace it. Spare sorting elements are incorporated in every level of the hierarchy and they not only can replace defective sorting elements in the corresponding level but also can be used to correct run-time errors. Detailed yield analysis is performed on the hierarchical sorting networks. Yield improvements for cases with various number of spares are evaluated. The simulation results show that the defect tolerant *HMSN* achieves a significant yield increase over a non-redundant sorting network.

In chapter 4, a new two-dimensional sorting algorithm, the *trapezoid sort* was presented. It is an improved algorithm over time complexity for sorting on two-dimensional SIMD arrays.

In addition, it preserves the properties of simple control hardware and ease of implementation of the *row-column sort*, and the complexity is improved to $\lceil \log_2 l \rceil + 1$ iterations with $l \approx \sqrt{n}$ for an $n \times n$ processor array. Like the "parallel bubble sort" of Sado and Igarashi and the "shear sort" of Scherson and Sen, the "trapezoid sort" is suboptimal for this architecture. However, the justification of the algorithm is on the simplicity of its data movements. The complicated data movements in mapping bitonic sort arise if $O(n)$ performance is attempted.

In chapter 5, several properties for the *trapezoid sort* were derived. The relationship between two rows in the array after each iteration of the *trapezoid sort* are derived based on the {0-1} principle. The relationship between the numbers of maximum dirty rows generated by input sequences with arbitrary number of zeros is also obtained. These results are then used to derive a more efficient method to find the k th-smallest value on mesh-connected processor arrays. The proposed method not only preserves the properties of the *row-column sort* such as simple control hardware and ease of implementation but also has less time complexity that approximately $\frac{3}{4}n$ processing steps are reduced in the first column-sort after the second iteration and the remaining steps will be reduced further by half after every successive iteration.

A novel *merge sort* method for the mesh-connected processor arrays was also presented in this chapter. Instead of an $O(m \log_2 m) T_N$ time complexity to sort mN input elements by the previous merge-split operation, only $O(\frac{m}{2}) T_N$ time complexity is sufficient to sort mN random inputs where T_N is the time complexity of the row-column sort algorithm. Therefore, we have achieved an $O(\log_2 m)$ order of improvement in time complexity to sort mN inputs. Other advantages of the proposed method include the simplicity of the architecture and efficient data movements with only near-neighbor communications. Therefore, it is very suitable for sorting

more than two sets of input elements in mesh-connected processor arrays.

7.2. Suggested Future Research

In this dissertation, I have concentrated on the developments of fault-tolerant VLSI systolic sorting arrays, defect-tolerant WSI sorting networks as well as sorting and merging algorithms on two-dimensional mesh connected processor arrays. Suggested future research issues include: (1) in addition to the analysis method, gate-level simulation can be performed to evaluate the actual fault coverage of the fault-tolerant systolic sorting array, (2) optimization in terms of the number of sorting elements in each level of the hierarchical modular sorting network may be generalized to a flexible *HMSN* which has more than three levels, and (3) in addition to using the merge-split method, the trapezoid sort algorithm can be further extended to k -dimensional mesh-connected processor arrays based on the modified odd-even merging algorithm.

APPENDIX A.

NETWORK TRANSFORMATION

Before we derive equivalent networks, some definitions based on those proposed by Wu and Feng [73] are introduced first. The physical names (or notations) for components inside an interconnection network T are defined as follows: (1) The stages in T are labeled from 0 to $l = \log_2 N - 1$. (2) The levels of links are labeled from 1 to l . (3) In a stage, each sorting element is denoted by binary bits $p_l \dots p_1$ representing its location in the stage and a link connected to the sorting element is represented by $p_l \dots p_1 p_0$ where $p_0 = 0$ for the link connected to the top input and $p_0 = 1$ for the link connected to the bottom input of the sorting element. The configuration of an interconnection network T is described by its describing rules.

THEOREM 1: A sorting block interconnected as a modified data manipulator is functionally equivalent to a sorting block interconnected as an *Omega* network.

PROOF: The topology equivalence between the *Omega* network and the modified data manipulator was shown in [73, 10] where the mapping function γ_i was derived as:

$$\gamma_i[(p_l p_{l-1} \dots p_1)_i] = (p_{l-i} \dots p_1 p_l p_{l-1} \dots p_{l-i+1})_i.$$

To further prove that these two networks are functionally equivalent, we exploit the property that if the input lines i and j from stage $k-1$ are processed by sorting element $(p_l \dots p_1)$ in stage k of the *Omega* network, they will be processed by $\gamma_k(p_l \dots p_1)$ for $k=0$ to $n-1$ in the modified data manipulator. The reason why k is from 0 to $l=n-1$ instead of from 0 to $l-1$ as for i , is that now we are considering the mapping of sorting elements but not the output links from it. Since

$$\gamma_k(p_l \dots p_1) = p_{l-k} \dots p_1 p_l \dots p_{l-k+1},$$

$$\begin{aligned} \gamma_0[(p_l p_{l-1} \dots p_1)] &= (p_l p_{l-1} \dots p_2 p_1), & \gamma_1[(p_l p_{l-1} \dots p_1)] &= (p_{l-1} p_{l-2} \dots p_1 p_l), \\ & \dots, \gamma_l[(p_l p_{l-1} \dots p_1)] &= (p_l p_{l-1} \dots p_2 p_1). \end{aligned}$$

The logical names of the input and output terminals in a modified data manipulator are the same as the physical names. This means the corresponding positions of the sorting elements in input and output terminals of the modified data manipulator are the same as those of the *Omega* network. □

APPENDIX B.

PERMUTATION TRANSFORMATION

Let a sequence $A = \{0, 1, \dots, 2^n-1\}$ ($N = 2^n$) be represented by $p_{n-1} \dots p_0$, and let τ, σ be two permutations of A .

DEFINITION 1: The shuffle permutation σ is defined as $\sigma : A \rightarrow A$ with $\sigma(p_{n-1} \dots p_0) = p_{n-2} p_{n-3} \dots p_1 p_0 p_{n-1}$. □

DEFINITION 2: The permutation τ is defined as $\tau : A \rightarrow A$ with $\tau(p_{n-1} \dots p_0) = p_{n-1} \dots p_0$ if $p_0 = 0$ and $\tau(p_{n-1} \dots p_0) = \overline{p_{n-1}} \overline{p_{n-2}} \dots \overline{p_1} p_0$ if $p_0 = 1$. □

DEFINITION 3: The Banyan permutation r is formed by setting all switching elements in the Banyan's interconnection network (this network can be viewed as a reverse network of the modified data manipulator in network topology) in straight connection states (see Fig. 3.5(b)). □

DEFINITION 4: The ψ permutation is formed by setting a switching element in the modified data manipulator either in straight or in exchange state. The switching elements in stage i ($i=0$ to $n-1$) with positions represented by $p_{n-1} \dots p_1$, will be in exchange state ($p_{n-1} \dots p_1 p_0 = p_{n-1} \dots p_1 \overline{p_0}$) if $p_{n-i} p_{n-i-1} = 01$ or 10 , or in straight state ($p_{n-1} \dots p_1 p_0 = p_{n-1} \dots p_1 p_0$) if $p_{n-i} p_{n-i-1} = 00$ or 11 . □

It should be noted that the topology describing rules only describe which switching elements in stage $i+1$ receive the outputs from switching elements in stage i . These rules do not describe whether the outputs should connect to the top input part or the bottom input part of a switching element. This is different from the permutation function which precisely describes

the link connections between two stages. Therefore, a permutation function has arguments from p_{n-1} to p_0 and a describing rule has arguments only from p_{n-1} to p_1 .

THEOREM 2: The shuffle permutation σ is topologically equivalent to the Banyan permutation r .

PROOF: Let N be the number of inputs and $p_{n-1} p_{n-2} \dots p_0$ represent links for each input, where $n = \log_2 N$. Also let $\sigma(p_{n-1} p_{n-2} \dots p_0)$ represent a shuffle permutation function applied to links $p_{n-1} \dots p_0$, and $r_i(p_{n-1} \dots p_0)$ represent the permutation function of the Banyan switching network at the i -th level ($1 \leq i < n-1$).

From the network topology we know that $\sigma(p_{n-1} \dots p_0) = p_{n-2} p_{n-3} \dots p_0 p_{n-1}$ and the permutations of the Banyan network can be described as $r_i(p_{n-1} \dots p_0) = p_{n-1} \dots p_{i+1} p_0 p_{i-1} \dots p_1 p_i$. We have

$$\begin{aligned}
 r_{n-1}(r_{n-2}(\dots(r_1(p_{n-1} p_{n-2} \dots p_0) \dots))) &= r_{n-1}(r_{n-2}(\dots(r_2(p_{n-1} p_{n-2} \dots p_2 p_0 p_1) \dots))) \\
 &= r_{n-1}(r_{n-2}(\dots(r_3(p_{n-1} \dots p_3 p_1 p_0 p_2) \dots))) \\
 &= \dots \\
 &= p_{n-2} p_{n-1} \dots p_0 p_{n-1} \\
 &= \sigma(p_{n-1} \dots p_0). \quad \square
 \end{aligned}$$

THEOREM 3: The τ permutation (Fig. 3.6(a)) is topologically equivalent to ψ permutation (Fig. 3.6(b)).

PROOF: Let N be the number of inputs, $p_{n-1} \dots p_1 p_0$ represent links for each input where $n = \log_2 N$, $\tau(p_{n-1} p_{n-2} \dots p_0)$ represent a τ permutation function applied to links $p_{n-1} \dots p_0$, and $\psi_i(p_{n-1} \dots p_0)$ represent the i -th level permutation function of the modified data manipulator.

From the network topologies we know that the permutation function for each network can be described as follows :

$\tau(p_{n-1} \dots p_0) = p_{n-1} p_{n-2} \dots p_0$ if $p_0=0$, and $\bar{p}_{n-1} \dots \bar{p}_1$ if $p_0=1$.

$\Psi_i(p_{n-1} \dots p_0) = p_{n-1} \dots p_{n-i+1} p_0 p_{n-i-1} \dots p_1 p_{n-i}$.

A switching element in stage i ($i=0$ to $n-1$), with position represented by $p_{n-1} \dots p_1$, will be either in exchange state ($p_{n-1} \dots p_1 p_0 = p_{n-1} \dots p_1 \bar{p}_0$) or in straight state ($p_{n-1} \dots p_1 p_0 = p_{n-1} \dots p_1 p_0$) depending on the switch control function $s_i(p_{n-1} \dots p_1 p_0)$. If $p_{n-i} p_{n-i-1} = 00$ or 11 , the corresponding element will be in the straight state, otherwise it will be in exchange state. For $i=0$ or $n-1$, these two functions will depend only on p_{n-1} and p_1 , respectively. We have

$$\begin{aligned}
 & s_{n-1}(\Psi_{n-1}(s_{n-2}(\Psi_{n-2}(\dots s_1(\Psi_1(s_0(p_{n-1} p_{n-2} \dots p_0) \dots)) \\
 & = s_{n-1}(\Psi_{n-1}(s_{n-2}(\Psi_{n-2}(\dots s_1(\Psi_1(p_{n-1,0} p_{n-2} \dots p_0) \dots)) + s_{n-1}(\Psi_{n-1}(s_{n-2}(\Psi_{n-2}(\dots \\
 & s_1(\Psi_1(p_{n-1,1} p_{n-2} \dots \bar{p}_0) \dots)) \\
 & = s_{n-1}(\Psi_{n-1}(s_{n-2}(\Psi_{n-2}(\dots s_1(p_0 p_{n-2} \dots p_{n-1}) \dots)) + s_{n-1}(\Psi_{n-1}(s_{n-2}(\Psi_{n-2}(\dots s_1(\bar{p}_0 p_{n-2} \dots \\
 & p_{n-1}) \dots)) \dots \\
 & = p_{0,0} p_{n-1,0} p_{n-2,0} \dots p_{3,0} p_{2,0} p_{1,0} + p_{0,1} \bar{p}_{n-1,0} \bar{p}_{n-2,0} \dots \bar{p}_{2,0} \bar{p}_{1,0} + p_{0,0} p_{n-1,0} \dots p_{3,0} \bar{p}_{2,0} \\
 & \bar{p}_{1,1} + p_{0,1} \bar{p}_{n-1,0} \bar{p}_{n-2,0} \dots \bar{p}_{3,0} p_{2,0} p_{1,1} + \dots + \bar{p}_{0,0} p_{n-1,1} p_{n-2,1} \dots p_{2,1} \bar{p}_{1,1} + p_{0,1} \bar{p}_{n-1,1} \\
 & \bar{p}_{n-2,1} \dots \bar{p}_{2,1} p_{1,1} \\
 & = \tau(p_{n-1} \dots p_0).
 \end{aligned}$$

The notation $p_{n-1,0} p_{n-2} \dots p_0$ means that a switch with location representation $p_{n-1}=0$ will be in bypass state, and $p_{n-1,1} p_{n-2} \dots \bar{p}_0$ means that a switch will be in exchange state if $p_{n-1}=1$.

□

REFERENCES

- [1] M. J. Foster and H. T. Kung, "The design of special-purpose VLSI chips," *Computer.*, vol. 13, pp. 26-40, Jan. 1980.
- [2] D. J. Kuck and R. A. Stokes, "The Burroughs scientific processor (BSP)," *IEEE Trans. Comput.*, vol. c-31, pp. 363-376, May 1982.
- [3] K. Hwang, P. -S. Tseng, and D. Kim, "An orthogonal multiprocessor for parallel scientific computations," *IEEE Trans. Comput.*, vol. c-38, pp. 47-60, Jan. 1989.
- [4] C. D. Thompson, "The VLSI complexity of sorting," *IEEE Trans. Comput.*, vol. c-32, pp. 1171-1184, Dec. 1983.
- [5] D. E. Knuth, *The art of computer programming - searching and sorting*. Reading, MA: Addison-Wesley, 1973.
- [6] K. E. Batcher, "Sorting networks and their applications," *Proc. AFIPS Conf.*, vol. 32, pp. 307-314, 1968.
- [7] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. c-20, pp. 153-161, Feb. 1971.
- [8] H. T. Kung, "Why systolic architectures?," *Computer*, vol. 15, pp. 37-46, Jan. 1982.
- [9] S. Horiguchi and Y. Shigei, "Wiring space complexity of systolic array," *Proc. Int. Workshop on Systolic Array*, pp. 1-10, 1986.
- [10] S. Horiguchi, "Systolic sorter for WSI implementation," *Int. Con. on Wafer Scale Integration*, pp. 151-160, 1989.
- [11] G. Bilardi and F. P. Preparata, "An architecture for bitonic sorting with optimal VLSI performance," *IEEE Trans. Comput.*, vol. c-33, pp. 646-651, July 1984.
- [12] D. A. Rennels, "Fault-tolerant computing - concepts and examples," *IEEE Trans. Comput.*, vol. c-33, pp. 1116-1129, Dec. 1984.
- [13] K. -L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Trans. Parallel and Distr. Sys.*, vol. Vol. 1, pp. 231-240, April 1990.
- [14] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Commun. Ass. Comput.*, vol. 20, pp. 263-271, Apr. 1977.
- [15] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Trans. Comput.*, vol. c-27, pp. 2-7, Jan 1979.
- [16] M. Kumar and D. S. Hirschberg, "An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes," *IEEE Trans. Comput.*, vol. c-32, pp. 254-264, Mar. 1983.
- [17] K. Hwang and F. A. Briggs, *Computer architecture and parallel processing*. McGraw Hill, 1984.

- [18] C. P. Schnorr and A. Shamir, "An optimal sorting algorithm for mesh-connected computer," *Proc. 18th ACM Symp. Theory Comput.*, pp. 255-261, Jan 1986.
- [19] K. Sado and Y. Igarashi, "Some parallel sorts on a mesh-connected processor array and their time efficiency," *Journal of Parallel and Distribut. Comput.*, vol. 3, pp. 398-410, 1986.
- [20] I. D. Scherson, S. Sen, and A. Shamir, "Shear sort: a true two-dimension sorting technique for VLSI networks," *International Conference on Parallel Processing*, pp. 903-908, 1986.
- [21] I. D. Scherson and S. Sen, "Parallel sorting in two-dimension VLSI models of computation," *IEEE Trans. Comput.*, vol. c-38, pp. 238-249, Feb. 1989.
- [22] U. Schwiegelshohn, "A shortperiodic two-dimensional systolic sorting algorithm," *IEEE International Conference on Systolic Arrays*, pp. 257-264, 1988.
- [23] Y. -H. Choi and M. Malek, "A fault-tolerant systolic sorter," *IEEE Trans. Comput.*, vol. c-37, pp. 621-624, May 1988.
- [24] C. E. Leiserson, "Systolic priority queues," *Proc. Caltech Conf. VLSI*, pp. 199-224, Jan. 1979.
- [25] A. C. Yao and F. F. Yao, "On fault-tolerant networks for sorting," *SIAM J. COMPUT.*, vol. 14, pp. 120-128, Feb. 1985.
- [26] L. Rudolph, "A robust sorting network," *IEEE Trans. Comput.*, vol. c-34, pp. 326-335, Apr. 1985.
- [27] J. Sun, J. Gecsei, and E. Cerny, "Fault-tolerance in balanced sorting networks," *Journal of Electronic Testing: Theory and Applications.*, vol. 1, pp. 31-41, 1990.
- [28] C. S. Kayvan, E. D. Goodman, and M. A. Shanblatt, "A concurrent error detection and correction algorithm for fault-tolerant VLSI arithmetic array processors," *Proc. Phoenix Conf. on Computer and Communication*, pp. 688-694, Feb. 1986.
- [29] R. J. Cosentino, "Concurrent error correction in systolic architectures," *IEEE Trans. CAD.*, vol. 7, pp. 117-125, Jan. 1988.
- [30] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Trans. Comput.*, vol. c-31, pp. 589-595, July 1982.
- [31] H. T. Kung and M. S. Lam, "Fault-tolerance and two level pipelining in VLSI systolic arrays," *MIT Conf. on Advanced Research in VLSI*, pp. 74-83, 1984.
- [32] K. Oflazer, "Design and implementation of a single-chip 1-D median filter," *IEEE Trans. Acoust. Speech, Signal Processing*, vol. Assp-28, pp. 1164-1168, Oct. 1983.
- [33] H. -W. Lang, M. Schimmler, H. Schemeck, and H. Schroder, "Systolic sorting on a mesh-connected network," *IEEE Trans. Comput.*, vol. c-34, pp. 652-658, July 1985.
- [34] S. H. Hosseini, "On fault-tolerant structure, distributed fault-diagnosis, reconfiguration, and recovery of the array processors," *IEEE Trans. Comput.*, vol. c-38, pp. 932-942, July 1989.
- [35] U. Manber, *Introduction to algorithms*. Reading, MA: Addison-Wesley, 1989.
- [36] J. J. Shedletsky, "Error correction by alternate data retry," *IEEE Trans. Comput.*, vol. C-25, pp. 106-117, Feb. 1978.
- [37] Y. -H. Choi and M. Malek, "A fault-tolerant FFT processor," *IEEE Trans. Comput.*, vol. c-37, pp. 617-621, May 1988.
- [38] B. W. Johnson, *Design and analysis of fault tolerant digital systems*. Reading, MA: Addison Wesley, 1989.

- [39] K. -H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. c-33, pp. 297-311, June 1984.
- [40] W. S. Song and B. R. Musicus, "A fault-tolerant architecture for a parallel digital signal processing machine," *Proc. International Conf. on Computer Design*, pp. 385-390, 1987.
- [41] J. -Y. Jou and J. A. Abraham, "Fault-tolerant FFT networks," *IEEE Trans. Comput.*, vol. c-37, pp. 297-311, May 1988.
- [42] S. Lin and D. J. Costello, *Error control coding: fundamental and applications*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
- [43] D. K. Pradhan and etc. and D. K. Pradhan and etc., and J. P. Hayes, *Computer architecture and organization*. Englewood Cliffs, NJ: McGraw Hill, 1984.
- [44] T. R. N. Rao and E. Fujiwara, *Error-control coding for computer systems*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
- [45] J. M. Berger, "A note on error detection codes for asymmetric channels," *Informat. Contr.*, vol. 4, pp. 68-73, Mar. 1961.
- [46] J. F. Wakery, *Error detecting codes, self-checking circuits and applications*. New York: North-Halland, 1978.
- [47] T. R. N. Rao, *Error coding for arithmetic processors*. New York: Academic Press, 1974.
- [48] H. Dong, "Modified Berger codes for detection of unidirection errors," *IEEE Trans. Comput.*, vol. c-33, pp. 572-575, June 1984.
- [49] W. C. Carter and P. R. Schneider, "Design of dynamically checked computers," *Proc. IFIP-68*, pp. 878-883, Aug. 1968.
- [50] M. A. Marouf and A. D. Friedman, "Design of self-checking checkers for Berger codes," *Proc. 8th Annu. Symp. on Fault-Tolerant Computing*, pp. 179-184, June 1978.
- [51] M. J. Ashjaee and S. M. Reddy, "On totally self-checking checkers for separable codes," *IEEE Trans. Comput.*, vol. c-26, pp. 737-744, Aug. 1977.
- [52] D. Nikolos, A. M. Paschalis, and G. Philokyprou, "Efficient design of totally self-checking checkers for all low-cost arithmetic codes," *IEEE Trans. Comput.*, vol. c-37, pp. 807-814, July 1988.
- [53] Z. Kohavi, *Switching and finite automata theory*. New York: McGraw-Hill Publishing Company, 1978.
- [54] S.-C. Liang and S.-Y. Kuo, "Fault-tolerant VLSI systolic median filters ," *Proc. Fourth Annu. Parallel Processing Symp.*, 1990.
- [55] L. R. Rabiner, M. R. Sambur, and C. E. Schmidt, "Applications of a nonlinear smoothing algorithm to speech processing," *IEEE Trans. Accoust., Speech, Signal Processing*, vol. ASSP-23, pp. 552-557, Dec. 1975.
- [56] E. Ataman, V. K. Atre, and K. M. Wong, "A fast method for real time median filtering," *IEEE Trans. Accoust. Speech, Signal Processing*, vol. Assp-28, pp. 415-421, Aug. 1980.
- [57] I. Pitas, "Fast algorithms for running order and max/min calculation," *IEEE Trans. Circuit and Systems*, vol. c-36, pp. 795-804, June 1989.
- [58] D. S. Richards, "VLSI median filters," *IEEE Trans. Accoust. Speech, Signal Processing*, vol. Assp-38, pp. 145-153, Jan. 1990.
- [59] D. Nicolas, J. Francis, S.-P. Marc, and D. Michel, "VLSI architecture for a one chip video median filter," *IEEE Int. Conf. Accoust., Speech and Signal Proc.*, vol. 3, pp. 1001-1004, 1985.

- [60] H. Schmeck, H. Schroder, and C. Strake, "Systolic s^2 -way merge sort is optimal," *IEEE Trans. Comput.*, vol. c-38, pp. 1052-1056, July 1989.
- [61] L. Snyder, "Introduction to the configurable, highly parallel computer," *IEEE Computer*, vol. 15, pp. 45-56, Jan. 1982.
- [62] R. Negrini, M. Sami, and R. Stefanelli, "Fault tolerance techniques for array structures used in supercomputing," *IEEE Computer*, pp. 78-87, Feb. 1986.
- [63] M. Dowd, Y. Perl, and M. Saks, "The balanced sorting network," *Proc. ACM Princ. Distrib. Comput.*, pp. 161-172, 1983.
- [64] N. Weste and K. Eshraghian, *Principles of CMOS VLSI design - a system perspective*. Reading, MA: Addison-Wesley, pp. 424-448.
- [65] K. Padmanabhan and D. H. Lawrie, "A class of redundant path multistage interconnection networks," *IEEE Trans. Comput.*, vol. c-32, pp. 1099-1108, Dec. 1983.
- [66] N.-F. Tzeng, P.-C. Yew, and C.-Q. Zhu, "A fault-tolerant scheme for multistage interconnection networks," *Sym. Comput. Architecture*, pp. 368-375, 1985.
- [67] D. K. Pradhan and etc. and D. K. Pradhan and etc., and J. P. Hayes, *Computer architecture and organization*. Englewood Cliffs, NJ: McGraw Hill, 1984.
- [68] A. Menn, G. B. Adams III, D. P. Agrawal, and H. J. Siegel, "Fault-tolerant multistage interconnection networks," *IEEE Computer*, pp. 14-27, June 1987.
- [69] M. S. Algudady, C. R. Das, and W. Lin, "Fault-tolerant task mapping algorithms for MIN-based multiprocessors," *Proc. International Conference on Parallel Processing*, 1990.
- [70] S.-C. Liang and S.-Y. Kuo, "Defect tolerant sorting networks for WSI implementation," *Int. Con. on Wafer Scale Integration*, pp. 131-137, 1990.
- [71] G. Bilardi, "Merging and sorting networks with the topology of the Omega network," *IEEE Trans. Comput.*, vol. c-38, pp. 1396-1403, Oct. 1989.
- [72] S. Horiguchi, "Fault tolerance performance of WSI systolic sorter," *Int. Con. on Wafer Scale Integration*, pp. 196-202, 1990.
- [73] C. L. Wu and T. Y. Feng, "On a class of multistage interconnection network," *IEEE Trans. Comput.*, vol. c-29, pp. 694-702, Aug. 1980.
- [74] S.-C. Liang and S.-Y. Kuo, "Concurrent error detection and correction in real-time systolic sorting arrays," *Proc. 20th Annu. Symp. on Fault-Tolerant Computing*, 1990.
- [75] M. Wang, M. Cutler, and S. Y. H. Su, "Reconfiguration of VLSI/WSI mesh array processors with two-level redundancy," *IEEE Trans. Comput.*, vol. c-38, pp. 547-554, April 1989.
- [76] C.H. Stapper, F.M. Armstrong, and K. Saji, "Integrated circuit yield statistics," *Proceedings of the IEEE*, vol. 71, pp. 453-470, April 1983.
- [77] P. Franzon, "Yield modeling for fault tolerant VLSI arrays," *The First Int. Workshop on Systolic Arrays*, pp. 207-216, July 1986.
- [78] T. E. Mangir and A. Avizienis, "Fault-tolerant design for VLSI: effect of interconnect requirements on yield improvement of VLSI designs," *IEEE Trans. Comput.*, vol. c-31, pp. 609-615, July 1982.
- [79] G. H. Barnes and et al. , "The ILLIAC IV computer," *IEEE Trans. Comput.*, vol. c-17, pp. 746-757, 1968.

- [80] S.-Y. Kuo and S.-C. Liang, "Efficient parallel sorting and merging algorithms for two-dimensional mesh-connected processor arrays," *will be published by International Conference on Parallel Processing*, 1991.
- [81] T. Leighton, "Tight bounds on the complexity of parallel sorting," *IEEE Trans. Comput.*, vol. c-34, pp. 344-354, Apr. 1985.
- [82] B. Parker and I. Parberry, "Constructing sorting networks from k -sorters," *Information Processing Letters*, vol. 33, pp. 157-162, Nov. 1989.
- [83] J. W. Tukey, *Exploratory data analysis*. Reading, MA: Addison-Wesley, 1977.
- [84] T. S. Huang, *Two-dimensional digital signal processing*. New York: Springer Verlag, 1981.
- [85] I. Scollar, B. Weidner, and T. S. Huang, "Image enhancement using the median and the interquartile distance," *Computer Vision, Graphics, and Image Processing*, vol. 25, pp. 236-251, 1984.
- [86] H. A. David, *Order statistics*. New York: Wiley, 1980.
- [87] I. Miller and J. E. Freund, *Probability and statistics for engineer*. Englewood Cliffs, NJ: Prentice-Hall, 1985.